FreeBSD™ JOURNAL

Vol. 1 • Issue No. 5

Science, Systems, and FreeBSD

ALSO

The ULE Scheduler

WHY You Should Set Up with PF!

LLDB in FreeBSD

(pfSense)^10

We're taking pfSense®
to a higher power.

Secure your network with pfSense®2.2 and FreeBSD®10 on the new Netgate®C2758.

With 8 Intel® processing cores, up to 8 GbE ports, AES-NI and multi-threaded packet filtering, we're advancing network security to a higher power.

Netgate®

7212 McNeil Drive Suite 204 Austin, TX 78729 +1.512.646.4100

Table of Contents

FreeBSD Journal
Vol. 1, Issue No. 5

Science, Systems, and FreeBSD

Keep FreeBSD Free!
(Donate Today...)

Your donatoins have helped FreeBSD by:

Funding Projects to Advance FreeBSD
Development Projects include: Updated video console (Newcons), UEFI system boot support, Native iSCSI kernel stack, New automounter, Capsicum improvements.

Hiring Additional Staff Members
The Foundation now has additional support with marketing and promotion.

Educating the Public and Promoting FreeBSD
We continue to provide funding and leadership through publishing the FreeBSD Journal, creating high-quality educational brochures, sponsoring conferences, and visiting companies to help facilitate collaboration efforts with the Project.

Providing Legal Support to the Project
You're helping us protect FreeBSD IP and provide legal guidance for the Project.

Building FreeBSD Project Infrastructure
Your funds allow for hardware purchases to keep the project growing.

What will your donation help the Foundation accomplish this year?

Support FreeBSD by donating to the FreeBSD Foundation

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system. Our mission is to continue and increase our support and funding to keep FreeBSD at the forefront of operating system technology.

Thanks to people like you, the FreeBSD Foundation has been proudly supporting the FreeBSD Project and community for over 14 years now. We are incredibly grateful for all the support we receive from you and so many individuals and organizations that value FreeBSD.

Make a gift to support our work throughout the rest of 2014. Help us continue and increase our support of the FreeBSD Project and community worldwide!

Making a donation is quick and easy. Go to:
www.freebsdfoundation.org/donate

The FreeBSD FOUNDATION

www.freebsdfoundation.org

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher •	Walter Andrzejewski walter@freebsdjournal.com
Editor-at-Large •	James Maurer jmaurer@freebsdjournal.com
Art Director •	Dianne M. Kischitz dianne@freebsdjournal.com
Office Administrator •	Cindy DeBeck cindy@freebsdjournal.com
Advertising Sales •	Walter Andrzejewski walter@freebsdjournal.com Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).

Published by the FreeBSD Foundation, PO Box 20247, Boulder, CO 80308
ph: 720-207-5142 • fax: 720-222-2350
e-mail: board@freebsdfoundation.org
Copyright © 2014 by FreeBSD Foundation. All rights reserved.

This magazine may not be reproduced in whole or in part without written permission from the publisher.

"Don't Hold Back!"

A few weeks ago, some *FreeBSD Journal* Editorial Board members comprised a portion of the attendees at EuroBSDcon, which took place in Sofia, Bulgaria, on September 27 and 28. We were really pleased to note that attendees not only knew about or subscribed to *FreeBSD Journal*, but also voiced some great ideas for new articles, some of which they've kindly volunteered to write.

I think we'd all agree that a Journal without fresh, relevant content would be pretty boring, so the fact that our concept of a Journal with articles focused specifically on FreeBSD has snowballed into offers of submissions at this relatively early stage of its development is not only a great compliment to the authors who've already graced these virtual pages, but also an indication that we're producing something unique and interesting enough to get people interested in writing for it.

If you've been reading the Journal, you already know that the writing quality is something we care about. The high quality that you've become accustomed to is partly in the original works, but it is also the result of the Journal's excellent editorial and design staff. Editors and designers work with all of the Journal's authors to get their writing, code examples, accompanying illustrations and article designs into the best shape possible. These articles are not only top-notch editorially, but they also look great. So if you have suggestions for articles, or your own outlines, abstracts, or even full article drafts that you think would be a good fit, contact us (editorial@freebsdjournal.com), and the Board will review them.

Returning to EuroBSDcon—this issue includes a brief conference report by Daniel Peyrolón, a 2014 Google Summer of Code student, who offers a taste of the conference for those who weren't able to attend. You'll also enjoy the Journal's recurring monthly columns: Ports Report (by Frederic Culot), svn update (by Glen Barber), and Dru Lavigne's This Month in BSD and Events Calendar.

You'll be pleased to see that this issue includes a diverse set of great articles, ranging from the latest debugger for FreeBSD (LLDB), to the use of FreeBSD in research, to the internals of the ULE scheduler, as well as a piece on the PF packet filter.

We're working on our next issue (Nov–Dec), the final issue of 2014, right now, and were busily planning for 2015. As I mentioned earlier, we want to know what you'd like to read about, so don't hold back! •

***FreeBSD Journal* Editorial Board**

Science, Systems, and FreeBSD™

by George Neville-Neil

The FreeBSD Operating System has its origins in academic research, having been derived from the 4.4-Lite version of the Berkeley Software Distribution developed by the Computer Systems Research Group (CSRG) at the University of California at Berkeley between the mid-1970s and the 1990s. The CSRG disbanded in 1995, after two decades of work in operating systems research [11]. In the two decades of active development, CSRG produced research on Virtual Memory [1], File Systems [12], produced the Sockets API and the most popular TCP/IP network stack as well as the Berkeley Packet Filter [10].

Although FreeBSD is now used heavily in industry, it continues to enjoy a great reputation in academia as a stable base on which to undertake research. There are many reasons why FreeBSD makes a great platform for research, including good tooling, consistent development and release processes, friendly open-source license, and a strong commitment to measurement.

The FreeBSD team places an emphasis on good tooling, including systems like DTrace, as well as using modern compilers such as LLVM and the associated LLDB debugger. FreeBSD follows a consistent set of development and release processes that allow researchers to know that their system—if built for a particular minor release such as 10.1—will work, unchanged on all the subsequent updates 10.2, 10.3, etc. This internal consistency within release branches reduces the number of variables that researchers need to worry about as they update their experiments and systems.

The BSD license allows researchers to use and repurpose parts of the operating system in their own work, and to then integrate that work into either research or products for industry without fear of being forced to give away their work. The license thereby encourages sharing without forcing it on the users of the system. The fact that FreeBSD is used in many commercial products gives researchers a large number of real-world scenarios in which to test their ideas—whether it is an improvement to the TCP software used by a CDN, or a new type of file system used to store zeta-bytes of data.

Finally, the FreeBSD Project has a commitment to measurement, always using data to back up any claims made about the system. The emphasis on measurement is a cultural value that was imparted by the original CSRG team and has remained current into the latest releases of the system. Current research efforts on FreeBSD continue to extend the system in several areas, including file systems, networking, and security.

File Systems

The Fast File System (FFS) is one of the longest-lived file-system development projects in computer science, having started in the early 1980s and continuing to the present day. FFS has a storied history of being surpassed by, and then surpassing, other storage research projects. In the early 1990s work on log-structured file systems (LFS), which store their data in a log rather than as a randomly accessible set of blocks on disk, was the subject of several research papers [17]. Performance comparisons were made between FFS and log-structured file systems showing that the log-structured systems had a significant performance advantage over more traditional file systems, like FFS. Following on the LFS work, research showed that the way FFS handled metadata—with information about the files stored on disk, rather than the data in the files—was the main source of the performance differences between the two systems. Research into soft updates, the ability to safely delay the writing of metadata to disk, improved the overall performance of FFS to the point where it met or exceeded the performance of log-based file systems [6].

The ability of file systems to make snapshots of their data, for use as fast, periodic backups, or as a way for ordinary users to recover files they had accidentally removed, was added to FFS in 1999.

With the coming of larger disks, exceeding a terabyte, the amount of time taken to recover disk state after a system crash grew from a few minutes to over an hour, to several hours in a large storage array. The well-known file-system check program (fsck) wound up blocking the system boot process while it checked to make sure that the disk state was consistent. Removing this system recovery bottleneck required the addition of a small journal to the soft update system [13]. Any pending updates would be kept in the journal, and only the journal, which rarely grows to more than a megabyte, would have to be read when a system restarted to recover a consistent disk state.

Work on FFS continues to the present day, with changes such as variable-sized disk blocks and enhanced metadata for files coming as part of a planned UFS3.

FreeBSD has adopted ZFS from OpenSolaris which promises to open up new areas of research with very large data sets. Although no research papers have been written about ZFS on FreeBSD, the ability to store extremely large, related sets of data on FreeBSD is important to data science researchers. As ZFS settles into FreeBSD as the file system of choice for massive data sets—those spanning hundreds of terabytes and more—it will surely spawn a new spate of research into efficient methods of data retrieval. While it is easy to store a great deal of data, for instance, by recording real-time measurements of physical phenomena, the focus of research is then on finding relevant data within these large data sets and making good use of them.

Networking

The networking subsystems in FreeBSD have always afforded platforms for research. The original sockets API and the canonical TCP/IP stack were both part of the original BSD system. One of the earliest platforms for TCP experiments was Dummynet, introduced into FreeBSD 2.2.8 in 1997 by Luigi Rizzo [15] from the University of Pisa. The purpose of Dummynet is to allow researchers working with new types of TCP congestion control algorithms to introduce network delay within a system in a test lab so that the researchers do not need to actually deploy their code on the open internet during their early tests. Dummynet introduces network delay by queuing packets internally and only releasing them at a preprogrammed time.

One of the major centers of TCP research is at Swinburne University, in Melbourne, Australia. Researchers at Swinburne, who study the efficiency of various TCP algorithms, developed a system to give them fine-grained visibility into the inner working of the TCP state machine and its tuning variables at run time. The SIFTR system, developed by Lawrence Stewart and James Healy, generates a run-time log of changes in the TCP state machine to help in evaluating new TCP algorithms [18].

As networking cards have gotten faster, surpassing first 10 gigabits per second and now reaching 40 Gbps, the operating system kernel itself has been seen as the bottleneck for networking applications. The netmap system provides a safe way to bypass the kernel's network stack to give applications direct access to the underlying networking hardware. With direct access to the underlying hardware, it is possible to have applications transmit and receive network data at wire speeds on commodity hardware [16].

Improving access to network packets has led to follow-on research, rewriting well-known applications such as web and name (DNS) servers. Ilias Marinos, a PhD student at the University of Cambridge, developed the Sandstorm and Namestorm systems, which leverage the netmap research to deliver static web content and DNS look ups at 10 Gbps on off-the-shelf hardware [8]. Using narrowly focused, purpose-built software, which foregoes generality for performance, the Sandstorm and Namestorm servers can fully exploit modern 10 and 40 Gbe hardware while using fewer CPU resources than the kernel's TCP/IP stack working through the sockets interface.

Security

The history of Unix spans the era of large, time-sharing systems. While the world of computing has switched from one in which multiple people shared a single computer into one in which any one person owns several computers, none of which is shared, the lessons learned in protecting users from each other were formative in terms of keeping security at the top of the list of features required in an operating system. Research into securing systems continues on FreeBSD with many of the leading ideas and first implementations showing up in FreeBSD releases.

The Mandatory Access Control framework, originally released with FreeBSD 5.0, provides fine-grained security policies to be implemented within an operating system kernel. The traditional security model for an operating system kernel usually has two levels of security: in the kernel, and everything else. Once a user or his code has gotten inside the kernel, he has the ability to get at anything else in the system. This coarse-grained type of security, while simple and easy to implement, restricts the types of secure systems that can be built. By bringing a finer-grained model of access controls to the kernel, the MAC framework makes it possible to build systems with narrower, well-defined security policies. The MAC framework moved from being an experimental feature in FreeBSD to being integrated into Mac OS, iOS and JunOS (the operating system in Juniper Routers) and continues to be the basis of research on security to this day [20, 19, 9].

One of the research systems built on top of the MAC framework is the Shill secure scripting language developed at Harvard University [14]. The Harvard group had the insight that one of the biggest problems in computer security is figuring out how to maintain and

ISILON The industry leader in Scale-Out Network Attached Storage (NAS)

Isilon is deeply invested in advancing FreeBSD performance and scalability. We are looking to hire and develop FreeBSD committers for kernel product development and to improve the Open Source Community.

We're Hiring!

With offices around the world, we likely have a job for you! Please visit our website at http://www.emc.com/careers or send direct inquiries to annie.romas@emc.com.

EMC² ISILON

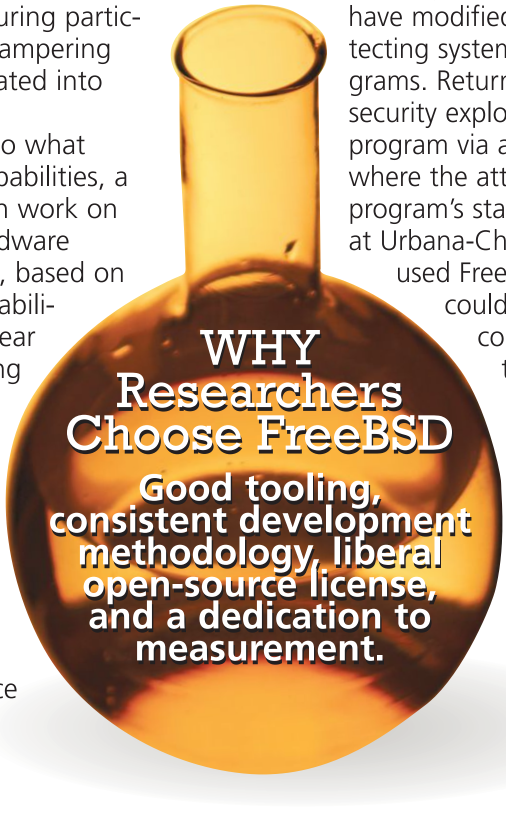
manage systems once they have been deployed. Typically, systems management software runs as "root," that is, with the all-powerful privilege that allows a user to do anything to the system with impunity. Most systems management code is written either in the shell scripting language or Perl. Their Shill programming language leverages the MAC framework to give shell-like scripts the same ability to express security policies as can be had in the kernel. Using Shill, the script programmer can allow access to some information without having to run the risk that an errant program will unintentionally overwrite or destroy data.

Early research into securing computer systems led to the study of "capabilities," unforgeable pieces of information that could be employed by applications used to prove that a user or a user's program was speaking truthfully [24]. At the time of the original research, capabilities were considered too expensive a mechanism to use for securing systems. Returning to this idea in 2009, Robert Watson at the University of Cambridge developed the Capsicum system with FreeBSD. Capsicum is an implementation of capabilities in the FreeBSD kernel that introduces minimal overhead compared to previous approaches. Programs running on a system with Capsicum can be sandboxed, isolating them from other applications, or developers can use the Capsicum APIs to create sandboxes within their programs, securing particularly sensitive data from tampering [21]. Capsicum was integrated into FreeBSD 10.0.

Probing more deeply into what could be achieved with capabilities, a larger research team began work on extending an available hardware instruction set architecture, based on the MIPS ISA, to bring capabilities down to the level of near zero overhead. The resulting CHERI and BERI (http://www.bericpu.org), open-source, hardware platforms, are now being used by researchers to investigate what can be done to enhance systems by modifying the hardware/software interface [5, 22, 23]. BERI is an open-source implementation in the BlueSpec language of a MIPS instruction set architecture that provides researchers with the ability to work at the hardware/software interface without having to build silicon from the ground up. CHERI builds upon the work done in BERI, adding support for a hardware-based capability system that uses specialized instructions added to the base MIPS ISA. These experimental systems are implemented in FPGAs for ease of use when doing iterative research, but if put into a dedicated CPU, would provide capabilities and their security primitives at the same speed as other native instructions.

FreeBSD was the first open-source project to adopt, wholesale, the LLVM compiler tool set. Originally developed at the University of Illinois at Urbana-Champaign, LLVM has been adopted as the compiler suite on Mac OS X and became the default compiler suite in FreeBSD 10 [7]. The adoption of LLVM was driven by several factors, but the main one was that LLVM is actually a toolkit for developing compilers and therefore more flexible in its application to various problems. Unlike the GNU compiler chain, which is designed against extension, LLVM is built such that targeting a new instruction set or application can be done quickly and easily. This ability to be retargeted to new applications has led to the use of LLVM in several research projects. A group of security researchers at the University of Illinois at Urbana-Champaign have modified LLVM to do research into protecting systems from return-oriented programs. Return-oriented programming is a security exploit that acquires control of a program via attacks such as stack smashing, where the attacker's code manipulates a program's stack during execution. The team at Urbana-Champaign extended LLVM and used FreeBSD to show that applications could still be protected from hostile code, even if the operating system on which they were running was compromised [4].

Using a different set of techniques, but still with the same research tools, the same research team showed that they could lock out return-oriented programs from attacking the operating system itself [3].


WHY
Researchers
Choose FreeBSD
Good tooling, consistent development methodology, liberal open-source license, and a dedication to measurement.

Training the Next Generation

A research platform is not a static system, but needs to constantly grow and change to remain relevant. To gather the best and brightest ideas, it is necessary to train researchers using the platform and then to have them adapt that platform to their needs and interests.

One of the features of FreeBSD that makes it amenable to research is the visibility provided into the system by various tools including hardware performance counters (hwpmc), which are used to measure system performance, and DTrace, which allows for the dynamic instrumentation of user and kernel level code at run time [2].

Using these tools, a team of researchers, developers, and educators, are preparing a masters level course in operating systems to be taught at the University of Cambridge starting in January 2015, and then shared with interested educators throughout academia. The ultimate goal of the course is to form the basis of a suite of courses covering all areas of operating system design and implementation.

Using tracing and profiling tools to teach an operating systems course is a radical departure from traditional teaching, which concentrates on discussing theoretical results and, when implementation is tried at all, adding trivial, but easy-to-grade, extensions to toy operating systems [25]. This new course will use a full production version of FreeBSD as its basic source code and will use the tools available on the platform to give students complete visibility into the inner workings of the operating system. Because the course is built around the availability of small and inexpensive embedded systems, which are well within the economic reach of average college students and computer science departments, each student will be able to work with a production operating system on a real hardware platform.

Once complete the teaching materials will be available, under an open-source license to other schools and educators through the http://www.teachbsd.org website.

RootBSD

Premier VPS Hosting

RootBSD has multiple datacenter locations, and offers friendly, knowledgeable support staff. Starting at just $20/mo you are granted access to the latest FreeBSD, full Root Access, and Private Cloud options.

www.rootbsd.net

Conclusion

Over the last 20 years the FreeBSD operating system has been an important contributor to both industrial and academic research with a strong track record of taking cutting-edge research and deploying it in commercial systems. Significant research contributions have been made in areas as diverse as file systems, networking, and security. Cutting-edge research continues to see the light of day first and foremost in FreeBSD. The reasons why researchers choose FreeBSD are as diverse as the work they undertake, but a few common themes are seen throughout, including good tooling, consistent development methodology, liberal open-source license, and a dedication to measurement. As the project continues to grow and evolve we can expect to see even more novel and cutting-edge research undertaken and published using FreeBSD. ●

George Neville-Neil works on networking and operating system code for fun and profit. He also teaches various courses on subjects related to computer programming. His professional areas of interest include code spelunking, operating systems, networking, and security. He is the coauthor with Marshall Kirk McKusick and Robert Watson of *The Design and Implementaion of the FreeBSD Operating System* and is the columnist behind ACM Queue magazine's *Kode Vicious*. Neville-Neil earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts. He is a member of the ACM, the Usenix Association, and the IEEE. He is an avid bicyclist and traveler and currently resides in New York City.

REFERENCES

[1] Babaoglu, Özalp and Joy, William. "Converting a Swap-based System to Do Paging in an Architecture Lacking Page-referenced Bits"; SOSP '81: *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*. (December 1981)

[2] Cantrill, Bryan; Shapiro, Michael; and Leventhal, Adam. "Dynamic Instrumentation of Production Systems"; *USENIX Annual Technical Conference*. (June 2007)

[3] Criswell, John; Dautenhahn, Nathan; and Adve, Vikram. "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels"; *SP '14: Proceedings of the 2014 IEEE Symposium on Security and Privacy. IEEE Computer Society*. (May 2014)

[4] Criswell, John; Dautenhahn, Nathan; and Adve, Vikram. "Virtual Ghost"; *ASPLOS 2014*. ACM Press. (2014)

[5] Davis, B; Norton, R.; Woodruff, J.; and Watson, R. N. M. "How FreeBSD Boots: A Soft-core MIPS Perspective." www.cl.cam.ac.uk.

[6] Ganger, Gregory R.; McKusick, Marshall Kirk; Soules, Craig A. N.; and Patt, Yale N. "Soft Updates: A Solution to the Metadata Update Problem in File Systems"; *Transactions on Computer Systems*. (May 2000)

[7] Lattner, Chris and Adve, Vikram. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation"; *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*: *Feedback-directed and Runtime Optimization*. IEEE Computer Society. (March 2004)

[8] Marinos, Ilias; Watson, Robert N. M.; and Handley, Mark. "Network Stack Specialization for Performance"; *Proceedings of the 2014 ACM Conference on SIG-COMM*. SIGCOMM '14. (2014)

[9] Mayer, Frank. "*Security Enhanced Linux Symposium-SELinux 2007*"; *Security Enhanced Linux Symposium-SELinux 2007*. (March 2007)

[10] McCanne, Steven and Jacobson, Van. "The BSD Packet Filter: A New Architecture for User-level Packet Capture"; *USENIX'93: Proceedings of the USENIX Winter 1993 Conference Proceedings*. USENIX Association. (January 1993)

[11] McKusick, Marshall Kirk. *Open Sources: Voices from the Open Source Revolution*. O'Reilly. (1999)

[12] McKusick, M. K.; Joy, W. N.; and Leffler, S. J. "A Fast File System for UNIX"; *ACM Transactions on Computing Systems*. (1984)

[13] McKusick, M. K. and Roberson , J. "Journaled Soft-Updates"; *Proceedings of EuroBSDCon*. (2010)

[14] Moore, S. "Shill: A Secure Shell Scripting Language"; *11th USENIX Symposium on Operating Systems Design and Implementation.* USENIX Association. (2014)

[15] Rizzo, Luigi. "Dummynet: A Simple Approach to the Evaluation of Network Protocols"; *SIGCOMM Computer Communication Review*. (January 1997)

[16] Rizzo, Luigi. "Netmap: A Novel Framework for Fast Packet i/o"; *2012 USENIX Annual Technical Conference*. USENIX Association. (2012)

[17] Rosenblum, Mendel and Ousterhout, John K. "The Design and Implementation of a Log-structured File System"; *ACM Transactions on Computer Systems*. (February 1992)

[18] Stewart , L. and Healy, J. "Characterizing the Behavior and Performance of SIFTR v1. 1.0."; http://caia.swin.edu.au/reports/070824A/CAIA-TR-070824A.pdf (2007)

[19] Watson, R.; Feldman, B.; Migus, A.; and Vance, C. *Design and Implementation of the Trusted BSD MAC framework*, Volume 1. IEEE. (2003)

[20] Watson, Robert N. M. "A Decade of OS Access-control Extensibility"; *Communications of the ACM*. (February 2013)

[21] Watson, Robert N. M.; Anderson, Jonathan; Laurie, Ben; and Kennaway, Kris. "Capsicum: Practical Capabilities for UNIX"; *USENIX Security'10: Proceedings of the 19th USENIX Conference on Security*. USENIX Association. (August 2010)

[22] Woodruff, J.; Watson, R. N. M.; Chisnall, D.; Moore, S. W.; Anderson, J.; Davis, B.; Laurie, B.; Neumann, P. G.; Norton, R.; and Roe, M. "The CHERI Capability Model: Revisiting RISC in an Age of Risk"; *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium*. (2014)

[23] Woodruff, Jonathan; Moore, Simon W.; and Watson, Robert N. M. "Memory Segmentation to Support Secure Applications"; *CEUR Workshop: Doctoral Symposium on Engineering Secure Software and Systems* (ESSoS). (January 2013)

[24] Dennis, J.B. and VanHorn, E.C. "Programming Semantics for Multi-programmed Computations," *Commun. ACM*, Vol. 9, No. 3, (1966). Also http://www.cs.virginia.edu/~evans/cs551/saltzer/

[25] https://www.usenix.org/conference/usenix-winter-1993-conference/presentation/nachos-instructional-operating-system. A reference to its child is: http://en.wikipedia.org/wiki/Pintos

FreeNAS

in an Enterprise Environment

By the time you're reading this, FreeNAS has been downloaded more than 5.5 million times. For home users, it's become an indispensable part of their daily lives, akin to the DVR. Meanwhile, all over the world, thousands of businesses universities, and government departments use FreeNAS to build effective storage solutions in myriad applications.

What you will learn...

- How TrueNAS builds off the strong points of the FreeBSD and FreeNAS operating systems
- How TrueNAS meets modern storage challenges for enterpri

The FreeNAS operating systems is freel the public and offers thorough docu active community, and a feature-rich the storage environment. Based on FreeBS can share over a host of protocols (SMB, FTP, iSCSI, etc) and features an intuitive the ZFS file system, a plug-in system fo much more.

Despite the massive popularity of aren't aware of its big brother dutif data in some of the most demandin environments: the proven, enterpri professionally-supported line of a

But what makes TrueNAS differe Well, I'm glad you asked...

Commercial Grade Suppo

When a mission critical storag organization's whole operati halt. Whole community-base free), it can't always get an and running in a timely ma responsiveness and expert dedicated support team provide that safety.

Created by the same developed FreeNAS.

NEW RELEASE

iX systems®

WE INTERRUPT THIS MAGAZINE TO BRING YOU THIS IMPORTANT ANNOUNCEMENT:

THE PEOPLE WHO DEVELOP FREENAS, THE WORLD'S MOST POPULAR STORAGE OS, HAVE JUST REVAMPED TRUENAS.

POWER WITHOUT CONTROL MEANS NOTHING. TRUENAS STORAGE GIVES YOU BOTH.

- ✓ Simple Management
- ✓ Hybrid Flash Acceleration
- ✓ Intelligent Compresssion
- ✓ All Features Provided Up Front (no hidden licensing fees)
- ✓ Self-Healing Filesystem
- ✓ High Availability
- ✓ Qualified for VMware and HyperV
- ✓ Works Great With Citrix XenServer®

To learn more, visit: www.iXsystems.com/truenas

POWERED BY INTEL® XEON® PROCESSORS

intel® Xeon® inside™

THE SEARCH FOR A NEW DEBUGGER

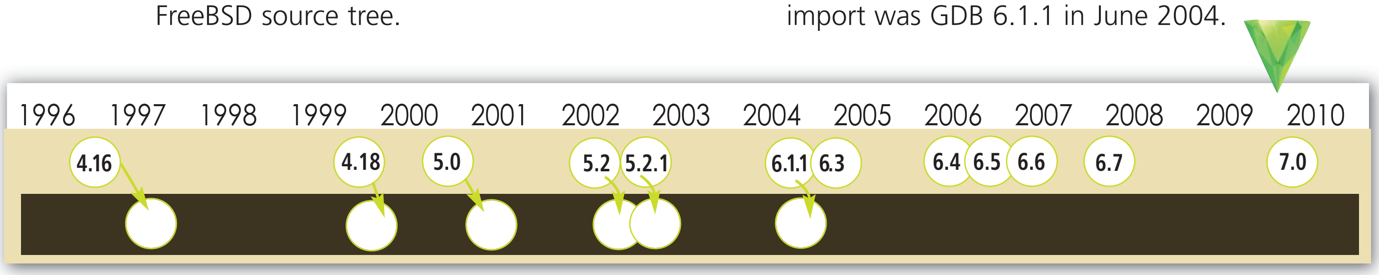
LLDB IN FreeBSD

by Ed Maste

One attribute that distinguishes FreeBSD from other open-source operating systems is the concept of the "base system," an integrated core that is developed, maintained, tested, and released as an integrated whole. Some major components of the base system are the kernel, userland libraries, system binaries, and development toolchain–and a key component of the toolchain is the debugger.

The GNU debugger, GDB, has long served as the base system's debugger. In 1993, the very first FreeBSD release included GDB 3.5. Every release since has included a version of GDB. The project followed GDB's development for more than a decade, and a number of different contributors incorporated new versions into the FreeBSD source tree.

This work produced a growing set of changes, and the effort increased with each new import. Project members attempted to have the changes incorporated to the upstream GDB project, but met resistance from the project's maintainers. Eventually this growing maintenance effort wore on the team, and the last import was GDB 6.1.1 in June 2004.



GPLv3

As with other GNU projects, in 2007 GDB's license changed to version 3 of the GNU Public License (GPLv3). The GPLv3 includes some restrictions that major FreeBSD contributors and consumers find unpalatable, and to date the project has avoided including any GPLv3-licensed code in the base system.

SEARCH FOR A NEW DEBUGGER

With a stale version of GDB in the base system and no clear path forward, it was clear FreeBSD needed a new debugger. Several open-source debugger projects were formed, both within and outside of the FreeBSD community. None of them reached critical mass to sustain development and produce a viable debugger we could use.

Then at their 2010 World-Wide Developer Conference, Apple announced they had their own debugger project, LLDB. It was released as open source in June of that year, and the next year it became the default debugger for Xcode, Apple's IDE. LLDB is provided under the University of Illinois/NCSA license which is a permissive, BSD-like license that is an ideal match for the FreeBSD project's licensing philosophy.

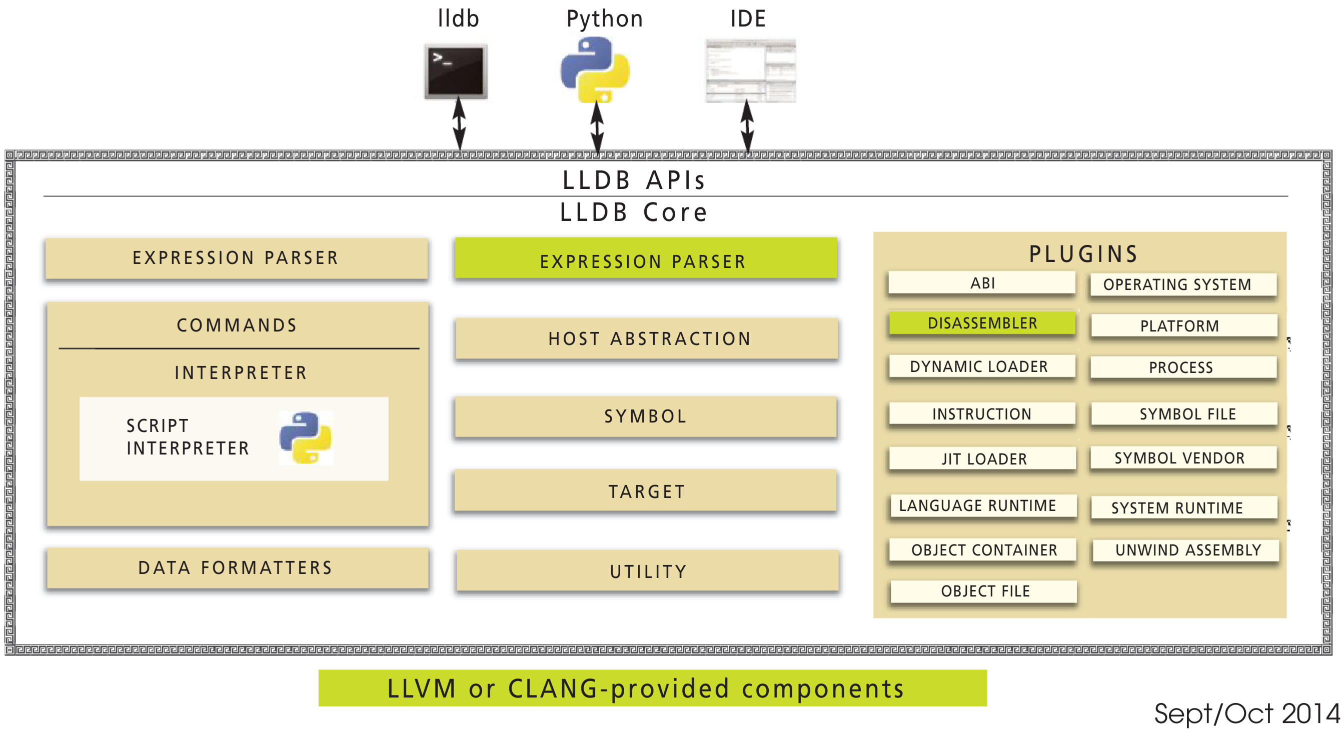
LLDB has since grown beyond an Apple project, with major contributions coming from open-source groups within companies like Intel and Google, and from FreeBSD, Debian, and other independent, open-source projects. Several people contributed to the FreeBSD port of LLDB, and our plan is that it will become the standard debugger in the FreeBSD base system.

LLDB DESIGN

LLDB is built as a set of modular components on top of the LLVM compiler infrastructure project and the Clang front-end. Reusing Clang and LLVM components allows for a great deal of functionality with lower effort and smaller code size compared to other debuggers. For example, LLDB has a full Clang compiler built in, used for its expression parser. If an expression is acceptable in a project's source code, it will also be handled by LLDB's expression parser, allowing a user to examine complex classes and data types in great detail and with confidence. LLVM provides LLDB's processor-specific support, including disassembly support and CPU-specific functionality. The modular design also provides the foundation for straightforward support of new processors, languages, and platforms.

A number of overall goals guide LLDB's design, and many of these derive from Clang and LLVM. To achieve high performance and reduce memory usage, LLDB attempts to parse only the debugging information required to perform an action. Threaded, high-performance classes provided by LLVM also contribute to LLDB's speed (see chart below).

LLDB aims to allow customization throughout so that a user can tailor the debugging experience. Variable and value display, type formatters, summary information, commands, prompts, and aliases can all be configured.



Like Clang and LLVM, LLDB inherently supports multiple CPU architectures and platforms within the same debugger binary. For example, the same debugger could be used to locally debug a native FreeBSD/amd64 application, examine a core file from a FreeBSD/MIPS system, and remotely debug an application running under a debug server on Linux. It's also possible to have multiple debug sessions running simultaneously in the same debugger.

As a debugger framework, LLDB is designed to be embedded in, or used by, other projects. IDEs and graphical front ends can easily incorporate and build on LLDB's functionality, using a C++ API. A full scripting API is also provided, currently supporting Python bindings. Python is usable from within LLDB, at the command line, for controlling execution after breakpoints, and for implementing new commands. The scripting interface is also available for external use; a Python script can create a debugger object and then use it to examine and control a debuggee's state, evaluate expressions, and so on.

LLDB USE

LLDB's command interpreter is designed with a consistent, structured syntax. Commands generally follow the pattern "noun verb"—for example "thread list" or "breakpoint set". The command syntax is somewhat more verbose than GDB, and adapting may take some effort for longtime GDB users. The benefit is that the command set is discoverable and regular; targeted autocompletion can provide relevant options to the user. As with GDB, commands may be abbreviated to the shortest unique prefix. An example of starting a debug session may look like:

```
% lldb
(lldb) target create /bin/ls
Current executable set to '/bin/ls' (x86_64).
(lldb) breakpoint set -name main
Breakpoint 1: where = ls`main + 33 at ls.c:163, address = 0x00000000004023f1
(lldb) process launch
```

LLDB has a powerful support for command aliases, and includes a built-in set of aliases for many GDB commands. Using the GDB aliases, the same result as above could be achieved with:

```
% lldb /bin/ls
Current executable set to '/bin/ls' (x86_64).
(lldb) b main
Breakpoint 1: where = ls`main + 33 at ls.c:163, address = 0x00000000004023f1
(lldb) run
```

The built-in aliases are limited though, and some of the more esoteric overloaded functionality provided by GDB commands is not available through the aliases. This is particularly true for breakpoints—in GDB the breakpoint command argument may be a line number, file name, function, or address, with sometimes overlapping or conflicting meaning. Migrating to LLDB's syntax and relying on the substring match to allow more concise commands is likely to be most effective.

Some GDB and LLDB Commands for Comparison

GDB	LLDB
Launch a process with no arguments (gdb) run (gdb) r	(lldb) process launch (lldb) run (lldb) r
Launch a process with arguments *args* (gdb) run *args* (gdb) r *args*	(lldb) process launch -- *args* (lldb) r *args*

(continues)

(Commands for Comparison continued)

GDB	LLDB
Launch a process in a new terminal window n/a	(lldb) process launch --tty -- ***args***
Attach to a process by pid (gdb) attach ***pid***	(lldb) process attach --pid ***pid*** (lldb) attach -p ***pid***
Source level single step (gdb) step (gdb) s	(lldb) thread step-in (lldb) step (lldb) s
Source level single step over function calls (gdb) next (gdb) n	(lldb) thread step-over (lldb) next (lldb) n
Source level single step out of current function (gdb) finish	(lldb) thread step-out (lldb) finish
Instruction level single step (gdb) stepi (gdb) si	(lldb) thread step-inst (lldb) si
Instruction level single step over function calls (gdb) nexti (gdb) ni	(lldb) thread step-inst-over (lldb) ni
Return immediately from current frame (gdb) return ***return expression***	(lldb) thread return ***return expression***
Set a breakpoint at all functions by name (gdb) break ***name*** (gdb) b ***name***	(lldb) breakpoint set --name ***name*** (lldb) br s -n ***name*** (lldb) b ***name***
Set a breakpoint by file and line (gdb) break ***file***:***line***	(lldb) breakpoint set --file ***file*** --line ***line*** (lldb) br s -f ***file*** -l ***line*** (lldb) b file:***line***
Set a breakpoint in all C++ methods by name (gdb) break ***name*** *assuming no C functions have the same name*	(lldb) breakpoint set --method ***name*** (lldb) br s -M ***name***
List breakpoints (gdb) info break	(lldb) breakpoint list (lldb) br l
Delete a breakpoint (gdb) delete 1	(lldb) breakpoint delete 1 (lldb) br del 1
Show arguments and local variables (gdb) info args *and* (gdb) info locals	(lldb) frame variable (lldb) fr v

The **"target create"** command (previous page chart) can create multiple targets in a debugging session with **"target list"** showing the active ones:

```
(lldb) target list
Current targets:
 target #0: /bin/app1 ( arch=x86-64-unknown-freebsd10.1, platform=host )
 target #1: /bin/app2 ( arch=x86-64-unknown-freebsd10.1, platform=host, pid=81166, state=exited )
```

To switch between them use **"target select <number>"**.

BREAKPOINTS

Breakpoints are set with the "breakpoint set" command. Breakpoints may be set at a given address, filename and line number, function or method name, or upon language-specific exceptions. Breakpoints may also be restricted to a specified thread or shared library.

Breakpoints are maintained as logical breakpoints within LLDB, which then resolve to one or more locations. The logical breakpoint and each resolved location are given integer identifiers, joined with a dot. For example, if the third breakpoint matches two function names and thus resolves to two locations, they will be called "3.1." and "3.2".

Breakpoints remain live throughout the debugging session, so loading a shared library with a function or method that matches an existing breakpoint specification results in new locations being added to that breakpoint. Similarly, unloading a shared library may remove breakpoint locations. A breakpoint remains set, but in an unresolved state after unloading all of its locations.

Whenever the debuggee process stops, LLDB prints relevant information: the thread that stopped, process location information including the address, filename, and line number, the current function and its arguments, and the stop reason. The reported stop reason may be a breakpoint, watchpoint, signal, address exception, or one of a number of target- or language-specific reasons. Finally, a small portion of the source code at the current address is shown.

```
* thread #1: tid = 100641, 0x00000000004023f1 ls`main(argc=1, argv=0x00007fffffffe760)
+ 33 at ls.c:163, name = 'ls', stop reason = breakpoint 1.1
    frame #0: 0x00000000004023f1 ls`main(argc=1, argv=0x00007fffffffe760) + 33 at ls.c:163
   160   #ifdef COLORLS
   161           char termcapbuf[1024];   /* termcap definition buffer */
   162           char tcapbuf[512];       /* capability buffer */
-> 163           char *bp = tcapbuf;
   164   #endif
   165
   166           (void)setlocale(LC_ALL, "");
(lldb)
```

EXAMINING DEBUGEE STATE

After encountering a breakpoint or stopping for another reason, LLDB selects the most relevant thread. This will be the one that encountered a breakpoint, received a signal, performed an invalid memory access, or otherwise triggered the stop.

The **"thread list"** command lists all active threads in the debuggee, with **"thread select"** choosing the desired thread for subsequent commands.

To obtain a stack backtrace use the **"thread backtrace"** command, also available with the **"bt"** alias. By default the current thread's backtrace is shown, but a different thread index may be provided as an argument, or **"all"** to show the stack for each thread.

While examining a backtrace the **"frame select"** command may be used to choose a specific frame. The **"up"** and **"down"** aliases provide short forms for relative frame selection. With a frame selected the "**frame variable**" command will display function arguments and local variables that are in scope.

CONTROLLING THE DEBUGGEE

LLDB groups the single-stepping process control commands under the top-level thread command. "**thread step-in**" steps a single source line, continuing into function calls. "**thread step-over**" also steps a single source line, but does not stop inside of a function call. "**thread step-out**" continues until the program returns from the current function. The "**thread until <line>**" command continues until the program reaches the specified source file line, or it returns from the current function.

The stepping commands have aliases to match GDB: "**s**" or "**step**" for thread step-in, "**n**" or "**next**" for thread step-over, and "**f**" or "**finish**" for thread step-out.

DATA FORMATTERS

LLDB has built-in support for a number of high-level data structure formats used by language runtimes and libraries. These take the internal representation of a variable and display it in a convenient user-facing format, as might be used in source code.

For FreeBSD, the C++ runtime library formatters are likely the most valuable. LLDB includes support for the two of interest in FreeBSD: libc++ and GNU libstdc++.

As an example, a **std::string** will show just the string contents by default, when the type formatter is enabled:

```
(lldb) expression str
(string) $1 = "This is a string."
```

Formatters may be disabled in order to see the full details of the data structure, if necessary:

```
lldb) type category disable gnu-libstdc++
(lldb) expression str
(string) $2 = {
  _M_dataplus = {_M_p  = "This is a string."}
}
```

LL
DB

SCRIPTING

LLDB's scripting interface may be accessed in multiple ways. The most basic is the "**script**" command, which invokes the embedded interpreter and may be used to query current program state through a set of convenience variables. Python may also be used to implement new LLDB commands.

LLDB can also invoke a script after hitting a breakpoint. The script can then control program state (for example, continuing the process), allowing for very complex breakpoint conditions.

Finally, LLDB can be used entirely from a Python script, without involving the stand-alone lldb binary. A script can "**import lldb**", create a debugger instance and then a target, set breakpoints, launch, single step, and continue the target, and examine variables or evaluate expressions.

LLDB IN FreeBSD ROADMAP

Ongoing development effort on the LLDB FreeBSD port takes place directly in the LLDB repository. It currently works well for the amd64 architecture, for both live and core file-based userland debugging. Core file debugging support also exists for the MIPS architecture. Between CPU support currently in progress by Google and work ongoing in the FreeBSD community, we expect to support 64- and

32-bit versions of the x86, ARM, MIPS, and PowerPC CPU architectures.

A Google Summer of Code (GSoC) 2014 project delivered a proof-of-concept for FreeBSD kernel debugging support; additional work is required to refine this before it can be integrated into LLDB.

One key component under development in the LLDB project is a remote debugging stub. This allows an LLDB instance running on one computer to access and control a process running on another. This is especially important for debugging on small embedded devices, which may lack the memory or computing power to accommodate a full-featured debugger. The debugging stub is initially being implemented for Linux, but the port to FreeBSD is relatively straightforward.

Snapshots of the LLDB tree are imported into FreeBSD-Current on an occasional basis. LLDB is not yet built by default, but may be enabled by adding `WITH_LLDB=yes` to `/etc/src.conf` before running "`make buildworld`" as described in the FreeBSD Handbook.

One complication in the FreeBSD base system is that it does not include Python, so the LLDB snapshot is currently built without scripting support. It remains functional, but as a result some of the more interesting and advanced features are not available. We are evaluating different approaches to address this, likely migrating the scripting interface to a run-time rather than compile-time option. This would allow all of LLDB's Python capabilities to be enabled by simply installing the Python package or port.

We expect to update Clang and LLVM in the FreeBSD base system to version 3.5 in the near future. LLDB will be updated at the same time, and we then expect to enable building it by default. ●

Ed Maste manages project development for the FreeBSD Foundation and works in an engineering support role with the University of Cambridge Computer Laboratory. He is also a member of the elected FreeBSD Core Team. Aside from FreeBSD and LLDB, he is a contributor to a number of other open-source projects, including QEMU and Open vSwitch. He lives in Kitchener, Canada, with his wife, Anna, and sons, Pieter and Daniel.

FreeBSD

VPS HOSTING in the Cloud

- Deploy in 30 Seconds
- 24/7/365 Live Support
- No Contract, Cancel Anytime
- Free Setup, Free DNS
- 100% Enterprise SSD
- 1TB Outbound Transfer
- 1 Gbps Port

Plans start at only 99¢ per month

atlantic.net℠

GET STARTED TODAY!

THE INTERNET NEEDS YOU

GET CERTIFIED AND GET IN THERE!

Go to the next level with BSD CERTIFICATION

Getting the most out of BSD operating systems requires a serious level of knowledge and expertise

NEED AN EDGE?

BSD Certification can make all the difference.

Today's Internet is complex. Companies need individuals with proven skills to work on some of the most advanced systems on the Net. With BSD Certification **YOU'LL HAVE WHAT IT TAKES!**

SHOW YOUR STUFF!

Your commitment and dedication to achieving the **BSD ASSOCIATE CERTIFICATION** can bring you to the attention of companies that need your skills.

BSDCERTIFICATION.ORG

Providing psychometrically valid, globally affordable exams in BSD Systems Administration

The FreeBSD ULE SCHEDULER

by Marshall Kirk McKusick and Jeff Roberson

THE GOAL of a multiprocessing system is to apply the power of multiple CPUs to a problem, or set of problems, to achieve a result in less time than it would run on a single-processor system. If a system has the same number of runnable threads as it does CPUs, then achieving this goal is easy. Each runnable thread gets a CPU to itself and runs to completion. Typically, there are many runnable threads competing for a few processors. One job of the scheduler is to ensure that the CPUs are always busy and are not wasting their cycles. When a thread completes its work, or is blocked waiting for resources, it is removed from the processor on which it was running. While a thread is running on a processor, it brings its working set—the instructions it is executing and the data on which it is operating—into the CPU's memory cache. Migrating a thread has a cost. When a thread is moved from one CPU to another, its CPU-cache working set is lost and must be removed from the CPU on which it was running and then loaded into the new CPU to which it has been migrated. The performance of a multiprocessing system with a naive scheduler that does not take this cost into account can fall beneath that of a single-processor system. The term ***processor affinity*** describes a scheduler that only migrates threads when necessary to give an idle processor something to do.

A multiprocessing system may be built with multiple processor chips. Each processor chip may have multiple CPU cores, each of which can execute a thread. The CPU cores on a single processor chip share many of the processor's resources, such as memory caches and access to main memory, so they are more tightly synchronized than the CPUs on other processor chips.

Handling processor chips with multiple CPUs is a derivative form of load balancing among CPUs on different chips. It is handled by maintaining a hierarchy of CPUs. The CPUs on the same chip are the cheapest between which to migrate threads. Next down in the hierarchy are processor chips on the same motherboard. Below them are chips connected by the same backplane. The scheduler supports an arbitrary depth hierarchy as dictated by the hardware. When the scheduler is deciding on which processor to migrate a thread, it will try to pick a new processor higher in the hierarchy because that is the lowest-cost migration path.

From a thread's perspective, it does not know that there are other threads running on the same processor, because the processor is handling them independently. The one piece of code in the system that needs to be aware of the multiple CPUs is the scheduling algorithm. In particular, the scheduler treats each CPU on a chip as one on which it is cheaper to migrate threads than it would be to migrate the thread to a CPU on another chip. The mechanism for getting tighter affinity between CPUs on the same processor chip versus CPUs on other processor chips is described later in this article.

The traditional FreeBSD scheduler maintains a global list of runnable threads that it traverses once per second to recalculate their priorities. The use of a single list for all runnable threads means that the performance of the scheduler is dependent on the number of tasks in the system, and as the number of tasks grows, more CPU time must be spent in the scheduler maintaining the list.

The ULE scheduler was developed during FreeBSD 5.0 with major work continuing into FreeBSD 9.0, spanning 10 years of development. The scheduler was developed to address shortcomings of the traditional BSD scheduler on multiprocessor systems. A new scheduler was undertaken for several reasons:

- To address the need for processor affinity in multiprocessor systems
- To supply equitable distribution of load between CPUs on multiprocessor systems
- To provide better support for processors with multiple CPU cores on a single chip
- To improve the performance of the scheduling algorithm so that it is no longer dependent on the number of threads in the system
- To provide interactivity and timesharing performance similar to the traditional BSD scheduler.

The traditional BSD scheduler had good interactivity on large timeshare systems and single-user desktop and laptop systems. However, it had a single global run queue and consequently a single global scheduler lock. Having a single global run queue was slowed both by contention for the global lock and by difficulties implementing CPU affinity.

The priority computation relied on a single global timer that iterated over every runnable thread in the system and evaluated its priority while holding several highly contended locks. This approach became slower as the number of runnable threads increased. While the priority calculations were being done, processes could not *fork()* or *exit()* and CPU s could not context switch.

The ULE Scheduler

The ULE scheduler can logically be thought of as two largely orthogonal sets of algorithms; those that manage the affinity and distribution of threads among CPUs and those that are responsible for the order and duration of a thread's runtime. These two sets of algorithms work in concert to provide a balance of low latency, high throughput, and good resource utilization. The remainder of the scheduler is event driven and uses these algorithms to implement various decisions according to changes in system state.

The goal of equaling the exceptional interactive behavior and throughput of the traditional BSD scheduler in a multiprocessor-friendly and constant-time implementation was the most challenging and time-consuming part of ULE's development. The interactivity, CPU utilization estimation, priority, and time slice algorithms together implement the timeshare scheduling policy.

The behavior of threads is evaluated by ULE on an event-driven basis to differentiate interactive and batch threads. Interactive threads are those that are thought to be waiting for and responding to user input. They require low latency to achieve a good user experience. Batch threads are those that tend to consume as much CPU as they are given and may be background jobs. A good example of the former is a text editor, and for the latter, a compiler. The scheduler must use imperfect heuristics to provide a gradient of behaviors based on a best guess of the category to which a given thread fits. This categorization may change frequently during the lifetime of a thread and must be responsive on timescales relevant to people using the system.

The algorithm that evaluates interactivity is called the interactivity score. The interactivity score is the ratio of voluntary sleep time to run time normalized to a number between 0 and 100. This score does not include time waiting on the run queue while the thread is not yet the highest priority thread in the queue. By requiring explicit voluntary sleeps, we can differentiate threads that are not running because of inferior priority versus those that are periodically waiting for user input. This requirement also makes it more challenging for a thread to be marked interactive as system load increases, which is desirable because it prevents the system from becoming swamped with interactive threads while keeping things like shells and simple text editors available to administrators. When plotted, the interactivity scores derived from a matrix of possible sleep and run times becomes a three-dimensional sigmoid function. Using this approach means that interactive tasks tend to stay interactive and batch tasks tend to stay batched.

A particular challenge is complex X Window applications such as Web browsers and office productivity packages. These applications may consume significant resources for brief periods of time; however the user expects them to remain interactive. To resolve this issue, a several-second history of the sleep and run behavior is kept and gradually decayed. Thus, the scheduler keeps a moving average that can tolerate bursts of behavior, but will quickly penalize timeshare threads that abuse their elevated status. A longer history allows longer bursts but learns more slowly.

The interactivity score is compared to the interactivity threshold, which is the cutoff point for considering a thread interactive. The interactivity threshold is modified by the process nice value. Positive nice values make it more challenging for a thread to be considered interactive, while negative values make it easier. Thus, the ***nice*** value gives the user some control over the primary mechanism of reducing thread-scheduling latency.

A thread is considered to be interactive if the ratio of its voluntary sleep time versus its run time is below a certain threshold. The interactivity threshold is defined in the ULE code and is not configurable. ULE uses two equations to compute the interactivity score of a thread. For threads whose sleep time exceeds their run time, Equation1 is used:

Eq. 1 $$\textit{interactivity score} = \frac{\textit{scaling factor}}{\textit{sleep / run}}$$

When a thread's run time exceeds its sleep time, Equation 2 is used instead:

Eq. 2 $$\textit{interactivity score} = \frac{\textit{scaling factor}}{\textit{run / sleep}} + \textit{scaling factor}$$

The scaling factor is the maximum interactivity score divided by two. Threads that score below the interactivity threshold are considered to be interactive; all others are noninteractive. The sched_interact_update() routine is called at several points in a thread's existence—for example, when the thread is awakened by a wakeup() call—to update the thread's run time and sleep time. The sleep- and run-time values are only allowed to grow to a certain limit. When the sum of the run time and sleep time passes the limit, they are reduced to bring them back into range. An interactive thread whose sleep history was not remembered at all would not remain interactive, resulting in a poor user experience. Remembering an interactive thread's sleep time for too long would allow the thread to get more than its fair share of the CPU. The amount of history that is kept and the interactivity threshold are the two values that most strongly influence a user's interactive experience on the system.

Priorities are assigned according to the thread's interactivity status. Interactive threads have a priority that is derived from the interactivity score and are placed in a priority band above batch threads. They are scheduled like real-time round-robin threads. Batch threads have their priorities determined by the estimated CPU utilization modified according to their process nice value. In both cases, the available priority range is equally divided among possible interactive scores or percent-cpu calculations, both of which are values between 0 and 100. Since there are fewer than 100 priorities available for each class, some values share priorities. Both computations roughly assign priorities according to a history of CPU utilization, but with different longevities and scaling factors.

ULE Implementation

The CPU utilization estimator accumulates run time as a thread runs and decays it as a thread sleeps. The utilization estimator provides the percent-cpu values displayed in ***top*** and ***ps***. ULE delays the decay until a thread wakes to avoid periodically scanning every thread in the system. Since this delay leaves values unchanged for the duration of sleeps, the values must also be decayed before any user process inspects them. This approach preserves the constant-time and event-driven nature of the scheduler.

The CPU utilization is recorded in the thread as the number of ticks during which a thread has been running, along with a window of time defined as a first and last tick (each tick is typically 1 millisecond). The scheduler attempts to keep roughly 10 seconds of history. To accomplish decay, it waits until there are 11 seconds of history and then subtracts one-tenth of the tick value while moving the first tick forward 1 second. This inexpensive, estimated moving-average algorithm has the property of allowing arbitrary update intervals. If the utilization information is inspected after more than the update interval has passed, the tick value is zeroed. Otherwise, the number of seconds that have passed divided by the update interval is subtracted.

The scheduler implements round-robin through the assignment of time slices. A time slice is a fixed interval of allowed run time before the scheduler will select another thread

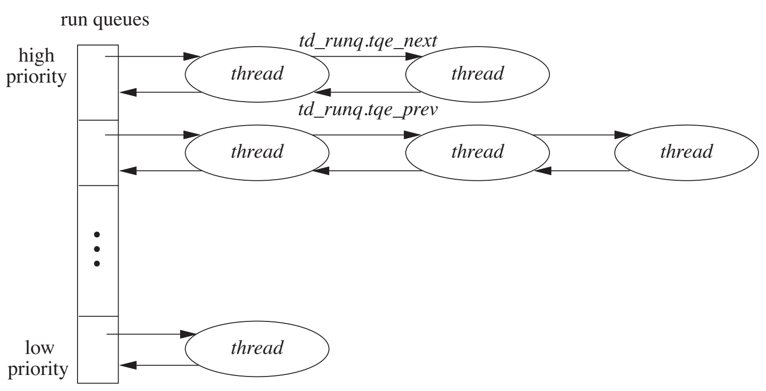


Fig. 1. Queueing structure for idle and real-time priority threads

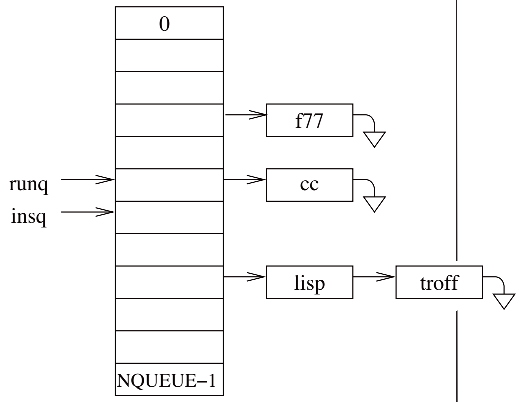


Fig. 2. A timeshare calendar queue

of equal priority to run. The time slice prevents starvation among equal priority threads. The time slice, times the number of runnable threads in a given priority, defines the maximum latency a thread of that priority will experience before it can run. To bound this latency, ULE dynamically adjusts the size of slices it dispenses based on system load. The time slice has a minimum value to prevent thrashing and balance throughput with latency. An interrupt handler calls the scheduler to evaluate the time slice during every statclock tick. Using the statclock to evaluate the time slice is a stochastic approach to slice accounting that is efficient but only grossly accurate.

The scheduler must also work to prevent starvation of low-priority batch jobs by higher-priority batch jobs. The traditional BSD scheduler avoided starvation by periodically iterating over all threads waiting on the run queue to elevate the low-priority threads and decrease the priority of higher-priority threads that had been monopolizing the CPU. This algorithm violates the desire to run in constant time independent of the number of system threads. As a result, the run queue for batch-policy timeshare threads is kept in a similar fashion to the system callwheel, also known as a calendar queue. A calendar queue is one in which the queue's head and tail rotate according to a clock or period. An element can be inserted into a calendar queue many positions away from the head and gradually migrate toward the head. Because this run queue is special purpose, it is kept separately from the real-time and idle queues while interactive threads are kept along with the real-time threads until they are no longer considered interactive.

The ULE scheduler creates a set of three arrays of queues for each CPU in the system. Having per-CPU queues makes it possible to implement processor affinity in a multiprocessor system.

One array of queues is the ***idle queue***, where all idle threads are stored. The array is arranged from highest to lowest priority. The second array of queues is designated the real-time queue. Like the idle queue, it is arranged from highest to lowest priority.

Figure 1 shows how the idle and real-time thread queues are organized as a doubly linked list of thread structures. The head of each run queue is kept in an array. Associated with this array is a bit vector, *rq_status*, that is used in identifying the nonempty run queues.

The third array of queues is designated the timeshare queue. Rather than being arranged in priority order, the timeshare queues are managed as a calendar queue as shown in Figure 2. The *runq* pointer references the current entry. The *runq* pointer is advanced once per system tick, although it may not advance on a tick until the currently selected queue is empty. When *runq* is incremented past the last queue, it is reset to point at the first queue. Since each thread is given a maximum time slice and no threads may be added to the current position, the queue will drain in a bounded amount of time. This requirement to empty the queue before advancing to the next queue means that the wait time a thread experiences is not only a function of its priority but also the system load.

The *insq* pointer references the base-point for insertion. Insertion into the timeshare queue is defined by the relative difference between a thread's priority and the best possible timeshare priority. When a thread becomes runnable or the currently running thread uses up its time slice, its position in the calendar queue is calculated using Equation 3:

Here *priority* is the thread's priority (adjusted based on its nice value) where small values rep-

$$\textit{queue index} = (\textit{insq\_index} + \textit{priority} - \textit{minimum\_batch\_priority}) \ \% \ \textit{NQUEUE} \qquad \text{Eq. 3}$$

resent high priorities and large values represent low priorities. The *insq* pointer is incremented every 10 milliseconds or any time that after runq is incremented *runq* and *insq* have the same value. High-priority threads will be placed soon after the current position. Low-priority threads will be placed far from the current position. This algorithm ensures that even the lowest-priority timeshare thread will eventually make it to the selected queue and execute in spite of higher-priority timeshare threads being available in other queues. The difference in priorities of two threads will determine their ratio of run time. The higher-priority thread may be inserted ahead of the lower-priority thread multiple times before the queue position catches up. This run-time ratio is what grants timeshare CPU hogs with different nice values, different proportional shares of the CPU.

These algorithms taken together determine the priorities and run times of timesharing threads. They implement a dynamic trade-off between latency and throughput based on system load, thread behavior, and a range of effects based on user-scheduling decisions made with nice. Many of the parameters governing the limits of these algorithms can be explored in real time with the `sysctl()` `kern.sched` tree. The rest are compile-time constants that are documented at the top of the scheduler source file `(/sys/kern/sched_ule.c)`.

Threads are picked to run, in priority order, from the realtime queue until it is empty, at which point threads from the currently selected timeshare queue will be run. Threads in the idle queues are run only when the other two arrays of queues are empty. Real-time and interrupt threads are always inserted into the real-time queues so that they will have the least possible scheduling latency. Interactive threads are also inserted into the real-time queue to keep the interactive response of the system acceptable.

Noninteractive threads are put into the timeshare queues and are scheduled to run when the queues are switched. Switching the queues guarantees that a thread gets to run at least once every pass around the array of the timeshare queues regardless of priority, thus ensuring fair sharing of the processor.

Multiprocessor Scheduling

A principal goal behind the development of ULE was improving performance on multiprocessor systems. Good multiprocessing performance involves balancing affinity with utilization and the preservation of the illusion of global scheduling in a system with local scheduling queues. These decisions are implemented using a CPU topology supplied by machine-dependent code that describes the relationships between CPUs in the system. The state is evaluated whenever a thread becomes runnable, a CPU idles, or a periodic task runs to rebalance the load. These events form the entirety of the multiprocessor-aware scheduling decisions.

The topology system was devised to identify which CPUs were symmetric multi-threading peers and then made generic to support other relationships. Some examples are CPUs within a package, CPUs sharing a layer of cache, CPUs that are local to a particular memory, or CPUs that share execution units such as in symmetric multi-threading. This topology is implemented as a tree of arbitrary depth where each level describes some shared resource with a cost value and a bitmask of CPUs sharing that resource. The root of the tree holds CPUs in a system with branches to each socket, then shared cache, shared functional unit, etc. Since the system is generic, it should be extensible to describe any future processor arrangement. There is no restriction on the depth of the tree or requirement that all levels are implemented.

Parsing this topology is a single recursive function called *cpu_search()*. It is a path-aware, goal-based, tree-traversal function that may be started from arbitrary subtrees. It may be asked to find the least- or most-loaded CPU that meets a given criteria, such as a priority or load threshold. When considering load, it will consider the load of the entire path, thus giving the potential for balancing sockets, caches, chips, etc. This function is used as the basis for all multiprocessing-related scheduling decisions. Typically, recursive functions are avoided in kernel programming because there is potential for stack exhaustion. However, the depth is fixed by the depth of the processor topology that typically does not exceed three.

When a thread becomes runnable as a result of a wakeup, unlock, thread creation, or other event, the *sched_pickcpu()* function is called to decide where it will run. ULE determines the best CPU based on the following criteria:

• Threads with hard affinity to a single CPU or short-term binding pick the only allowed CPU.

• Interrupt threads that are being scheduled by their hardware interrupt handlers are scheduled on the current CPU if their priority is high enough to run immediately.

• Thread affinity is evaluated by walking backwards up the tree starting from the last CPU on which it was scheduled until a package or CPU is found with valid affinity that can run the thread immediately.

• The whole system is searched for the least-loaded CPU that is running a lower-priority thread than the one to be scheduled.

• The whole system is searched for the least-loaded CPU.

• The results of these searches are compared to the current CPU to see if that would give a preferable decision to improve locality among the sleeping and waking threads as they may share some state.

This approach orders from most preferential to least preferential. The affinity is valid if the sleep time of the thread was shorter than the product of a time constant and a largest-cache-shared level in the topology. This computation coarsely models the time required to push state out of the cache. Each thread has a bitmap of allowed CPUs that is manipulated by ***cpuset*** and is passed to *cpu_search()* for every decision. The locality between sleeper and waker can improve producer/consumer type threading situations when they have shared cache state but it can also cause underutilization when each thread would run faster given its own CPU. These examples exemplify the types of decisions that must be made with imperfect information.

The next major multiprocessing algorithm runs when a CPU idles. The CPU sets a bit in a bitmask shared by all processors that says that it is idle. The idle CPU calls *tdq_idled()* to search other CPUs for work that can be migrated, or stolen in ULE terms, to keep the CPU busy. To avoid thrashing and excessive migration, the kernel sets a load threshold that must be exceeded on another CPU before some load will be taken. If any CPU exceeds this threshold, the idle CPU will search its run queues for work to migrate. The highest-priority work that can be scheduled on the idle CPU is then taken. This migration may be detrimental to affinity but improves many latency-sensitive workloads.

Work may also be pushed to an idle CPU. Whenever an active CPU is about to add work to its own run queue, it first checks to see if it has excess work and if another CPU in the system is idle. If an idle CPU is found, then the thread is migrated to the idle CPU using an ***interprocessor interrupt (IPI).*** Making a migration decision by inspecting a shared bitmask is much faster than scanning the run queues of all the other processors. Seeking out idle processors when adding a new task works well because it spreads the load when it is presented to the system.

The last major multiprocessing algorithm is the long-term load balancer. This form of migration, called ***push migration***, is done by the system on a periodic basis and more aggressively offloads work to other processors in the system. Since the two scheduling events that distribute load only run when a thread is added and when a CPU idles, it is possible to have a long-term imbalance where more threads are running on one CPU than another. Push migration ensures fairness among the runnable threads. For example, with three runnable threads on a two-processor system, it would be unfair for one thread to get a processor to itself while the other two had to share the second processor. To fulfill the goal of emulating a fair global run queue, ULE must periodically shuffle threads to keep the system balanced. By pushing a thread from the processor with two threads to the processor with one thread, no single thread would get to run alone indefinitely. An ideal implementation would give each thread an average of 66 percent of the CPU available from a single CPU.

The long-term load balancer balances the worst path pair in the hierarchy to avoid socket-, cache-, and chip-level imbalances. It runs from an interrupt handler in a randomized interval of roughly 1 second. The interval is randomized to prevent harmonic relationships between periodic threads and the periodic load balancer. In much the same way a stochastic sampling profiler works, the balancer picks the most- and least-loaded path from the current tree position and then recursively balances those paths by migrating threads.

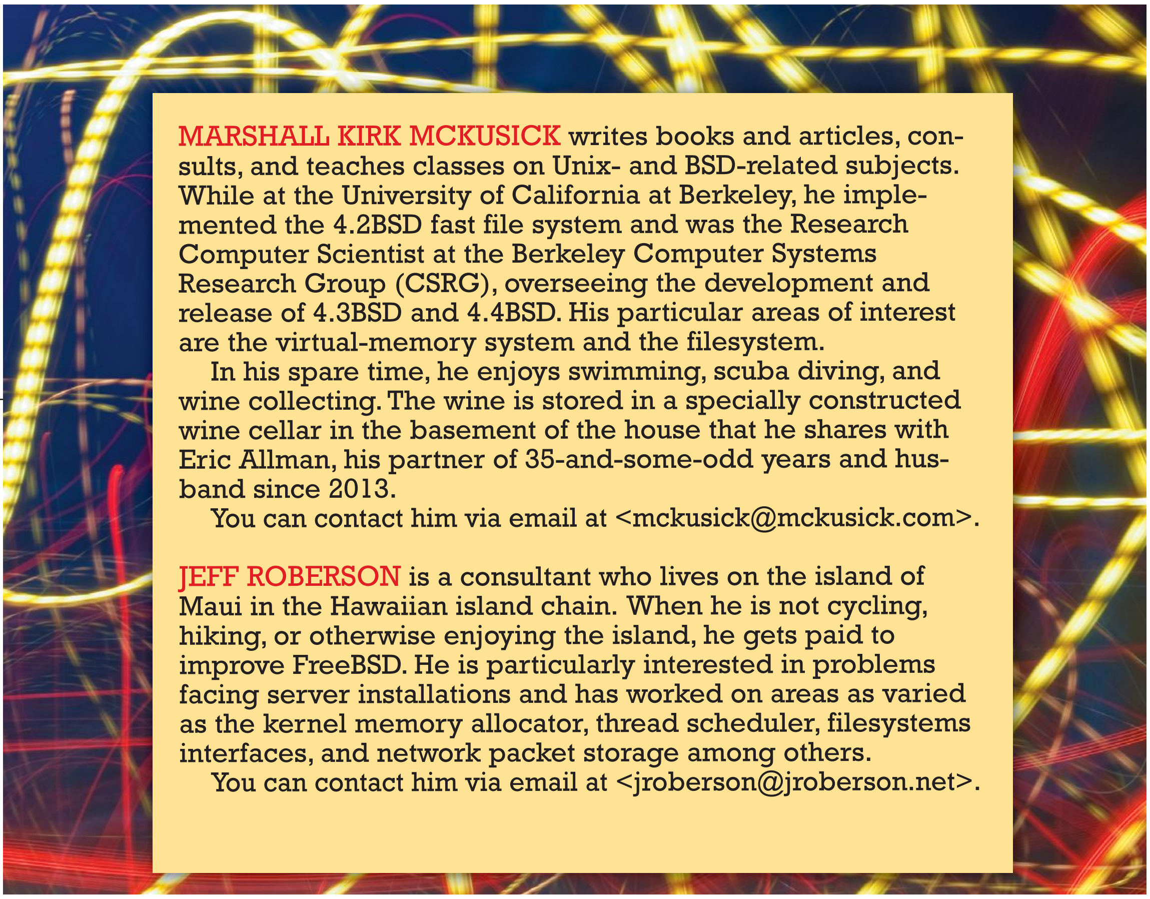
The scheduler must decide whether it is necessary to send an IPI when adding a thread to a remote CPU, just as it must decide whether adding a thread to the current CPU should preempt the current thread. The decision is made based on the current priority of the thread running on the target CPU and the priority of the thread being scheduled. Preempting whenever the pushed thread has a higher priority than the currently running thread results in excessive interrupts and preemptions. Thus, a thread must exceed the timesharing priority before an IPI is generated. This requirement trades some latency in batch jobs for improved performance.

A notable omission to the load balancing events is thread preemption. Preempted threads are simply added back to the run queue of the current CPU. An additional load-balancing decision can be made here. However, the run-time of the preempting thread is not known and the preempted thread may maintain affinity. The scheduler optimistically chooses to wait and assume affinity is more valuable than latency.

Each CPU in the system has its own set of run queues, statistics, and a lock to protect these fields in a `thread-queue` structure. During migration or a remote wakeup, a lock may be acquired by a CPU other than the one owning the queue. In practice, contention on these locks is rare unless the workload exhibits grossly overactive context switching and thread migration, typically suggesting a higher-level problem. Whenever a pair of these locks is required, such as for load balancing, a special function locks the pair with a defined lock order. The lock order is the lock with the lowest pointer value first. These per-CPU locks and queues resulted in nearly linear scaling with well-behaved workloads in cases where performance previously did not improve with the addition of new CPUs and occasionally have decreased as new CPUs introduced more contention. The design has scaled well from single CPUs to 512-thread network processors.

Adaptive Idle

Many workloads feature frequent interrupts that do little work but need low latency. These workloads are common in low-throughput, high-packet-rate networking. For these workloads, the cost of waking the CPU from a low-power state, possibly with an IPI from another CPU, is excessive. To improve performance, ULE includes a feature that optimistically spins, waiting for load when the CPU has been context switching at a rate exceeding a set frequency. When this frequency lowers or we exceed the adaptive spin count, the CPU is put into a deeper sleep. ●



MARSHALL KIRK MCKUSICK writes books and articles, consults, and teaches classes on Unix- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast file system and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG), overseeing the development and release of 4.3BSD and 4.4BSD. His particular areas of interest are the virtual-memory system and the filesystem.

In his spare time, he enjoys swimming, scuba diving, and wine collecting. The wine is stored in a specially constructed wine cellar in the basement of the house that he shares with Eric Allman, his partner of 35-and-some-odd years and husband since 2013.

You can contact him via email at <mckusick@mckusick.com>.

JEFF ROBERSON is a consultant who lives on the island of Maui in the Hawaiian island chain. When he is not cycling, hiking, or otherwise enjoying the island, he gets paid to improve FreeBSD. He is particularly interested in problems facing server installations and has worked on areas as varied as the kernel memory allocator, thread scheduler, filesystems interfaces, and network packet storage among others.

You can contact him via email at <jroberson@jroberson.net>.

This article is based on the discussion of the ULE scheduler in Chapter 4 of ***The Design and Implementation of the FreeBSD Operating System, Second Edition****. The material from that chapter is used with the permission of Pearson Education.*

Testers, Systems Administrators, Authors, Advocates, *and of course* **Programmers** *to join any of our diverse teams.*

COME JOIN THE PROJECT THAT MAKES THE INTERNET GO!

★ DOWNLOAD OUR SOFTWARE ★

http://www.freebsd.org/where.html

★ JOIN OUR MAILING LISTS ★

http://www.freebsd.org/community/mailinglists.html?

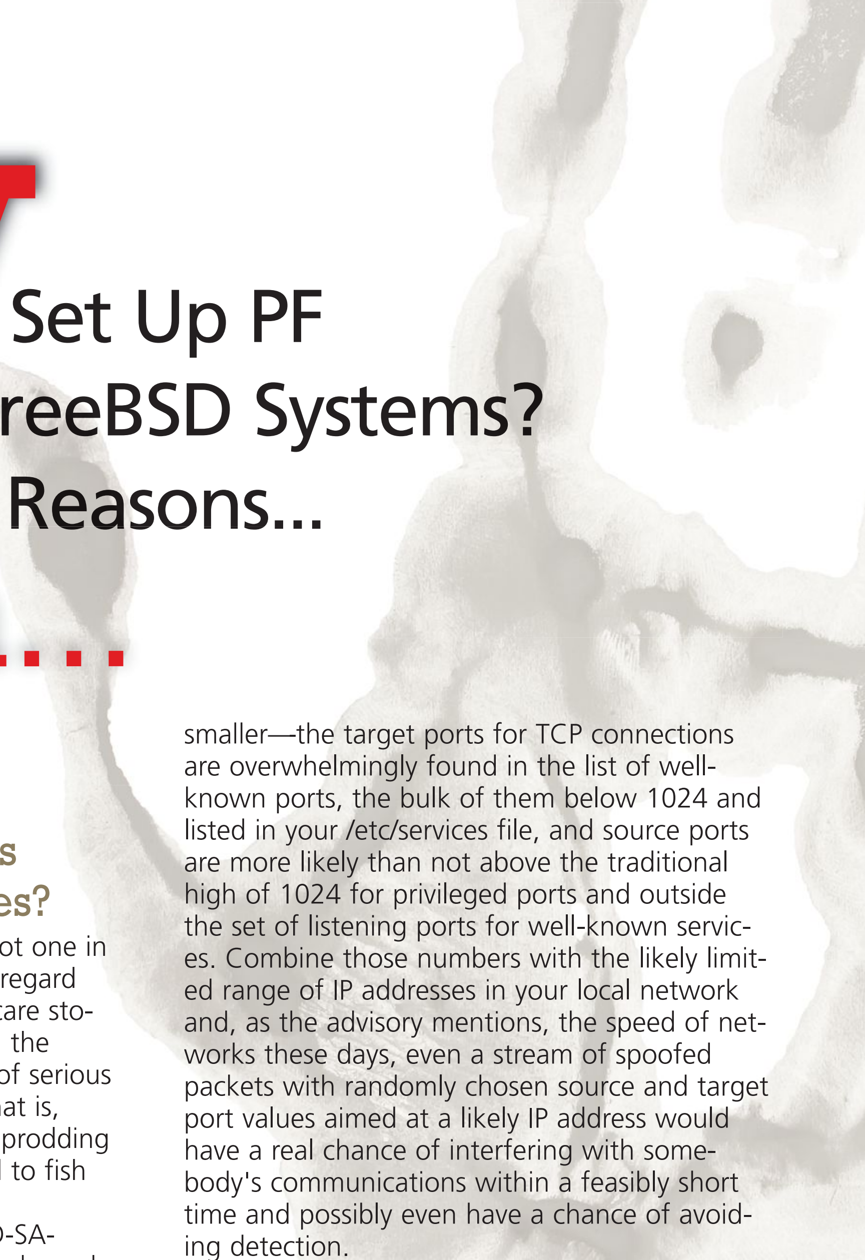
★ ATTEND A CONFERENCE ★

• bsdcan.org • asiabsdcon.org

The FreeBSD Project

The FreeBSD FOUNDATION

why Set Up PF on Your FreeBSD Systems? A Handful of Reasons... Read On...



by Peter N. M. Hansteen

Remember This Summer's Trouble with Window Sizes?

The summer of 2014 was a long and hot one in the Northern Hemisphere, but if we disregard for now the stream of advisories and scare stories involving OpenSSL-originated code, the summer has also been reasonably free of serious security issues that impact the BSDs. That is, until the very day I noticed the friendly prodding for this FreeBSD-related article I needed to fish out from my general backlog.

The security advisory dubbed FreeBSD-SA-14:19.tcp is at heart just another missing bounds check—a check for whether a particular bit of arriving data lies within the expected range. The data in this instance are sequence numbers for packets belonging to an already established TCP connection, which should reasonably be expected to match an anticipated interval, or window, of allowed values. In some cases a missing bounds check would be a "meh" at best, or somewhat embarrassing at worst, but with the code sitting deep in the network stack, the possible consequences were predictably serious: With a correctly guessed set of endpoint addresses and port numbers, an attacker would be able to tear down a connection with a crafted TCP SYN from a spoofed address.

The actual odds of success are slight, but not overwhelmingly bad: TCP ports are 16-bit numbers, and guessing correctly for both sides would double the list of possible values to a mere 17 bits or 131072 in decimal. In real life, however, the set of possible values is quite a bit smaller—the target ports for TCP connections are overwhelmingly found in the list of well-known ports, the bulk of them below 1024 and listed in your /etc/services file, and source ports are more likely than not above the traditional high of 1024 for privileged ports and outside the set of listening ports for well-known services. Combine those numbers with the likely limited range of IP addresses in your local network and, as the advisory mentions, the speed of networks these days, even a stream of spoofed packets with randomly chosen source and target port values aimed at a likely IP address would have a real chance of interfering with somebody's communications within a feasibly short time and possibly even have a chance of avoiding detection.

Now please go ahead and use your favorite tool for upgrading your system to a non-vulnerable version. It's always useful to keep up with bug fixes and apply patches as soon as practical after they become available. But the fact remains (and now I'm trying very hard not to sound like a condescending OpenBSD zealot) that with PF enabled on your FreeBSD system, and even a four byte `/etc/pf.conf`—sole content being pass, which is again very close to the default `pf.conf`. OpenBSD now ships in the default installation—you would have been protected against any attacker who tried to exploit the missing bounds check bug. That's due to one of the basic features of any stateful firewall—packets with sequence numbers outside the expected range are simply discarded, and the out-of-range SYN would never have made it to anywhere that mattered.

Bugs should be fixed of course, but I count this bug as one data point in why having a stateful packet filter enabled on your network, somewhere in the signal path between any probable attacker and your system, is a useful thing.

Stateful Trouble Removal – The Four-Byte Minimum

FreeBSD comes with PF in the kernel and the basic tools in the base system, but it is not enabled by default, and the last time I checked, the system does not supply a default `pf.conf` either, so just enabling PF using the supplied rc scripts will fail until you supply a valid `pf.conf`.

Now that you know that a `pf.conf` that contains only the line

```
pass
```

will be enough, why not start with that?

If you try reloading the rule in that minimalistic file and apply the -v (verbose) option to `pfctl`, you will see the rule like it is actually loaded:

```
$ sudo pfctl -vf /etc/pf.conf
pass all flags S/SA keep state
```

Since OpenBSD 4.1 and FreeBSD 7, these are the default flags and state options. You can set other combinations explicitly if you like, but this set of sane defaults makes new connections that match a pass rule create state with sane options, and as we already mentioned, even this basic and quite open default serves to protect your systems from some kinds of mischief. In fact, this four byte `pf.conf` could be useful even on systems that do not act as gateways or serve Internet-facing services.

To Absolute Security and Beyond

And talking of mischief, you have probably noticed that password guessing by robots aimed at a number of services is an almost constant source of log noise, or worse, that has if anything not become better over the years. If you run services on the Internet or other potentially noisy networks, you're probably interested in ways to reduce the amount of noise you have to deal with. A few simple PF rules can take you a long way towards that goal too.

But first, let us go for the five-byte maximum security rule set -

```
block
```

which in turn loads like this

```
$ sudo pfctl -vf pf.conf
block drop all
```

Even without my help, you can probably deduce that this particular rule will not let anything pass. No traffic passes. A network doorstop if there ever was one.

Now Expand to Allow Some Useful Traffic

I have yet to see a system actually run with this as its complete rule set for more than a few minutes, but it is in fact a very useful starting point. If you start with a default to block, you allow only those services you know you need, and you will end up with a setup where you have far better control of what passes in and out of your systems than with a more permissive default. I have seen a number of systems that have precisely that (block) as its first rule, with other rules following that let specific kinds of traffic through. This reveals indirectly the structure of PF rule sets: the last matching rule wins. In a typical simple configuration, a rule set like this:

```
clients = "192.168.103/24"
backupserver = "192.0.2.227"
bacula_ports = "9101:9103"
tcp_ports = "{ ftp, ssh, domain, ntp, whois, www, https, auth, nntp, imaps, \
                        rtsp, submission 8080:8082 }"
udp_ports = "{domain, ntp}"
block
pass inet proto tcp from $clients to port $tcp_ports
pass inet proto udp from $clients to port $udp_ports
pass inet proto tcp from $backupserver to $clients port $bacula_ports
```

- the logic is quite clear. The initial block rule stops everything by default, while the next three rules all make specific traffic from specific hosts pass, as long as it's aimed at the specific ports given. OK, I jumped ahead a bit and introduced a couple of other features too, macros and lists. Macros expand in-place, and lists like the macros `tcp_ports` and `udp_ports` make the parser generate one rule per list item, so the rules that are actually loaded look more like this: (next page)

```
$ sudo pfctl -vf /etc/pf.conf
clients = "192.168.103/24"
backupserver = "192.0.2.227"
bacula_ports = "9101:9103"
tcp_ports = "{ ftp, ssh, domain, ntp, whois, www, https, auth, nntp, imaps, rtsp, submission 8080:8082 }"
udp_ports = "{domain, ntp}"
block drop all
pass inet proto tcp from 192.168.103.0/24 to any port = ftp flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = ssh flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = domain flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = ntp flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = nicname flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = http flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = https flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = auth flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = nntp flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = imaps flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = rtsp flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port = submission flags S/SA keep state
pass inet proto tcp from 192.168.103.0/24 to any port 8080:8082 flags S/SA keep state
pass inet proto udp from 192.168.103.0/24 to any port = domain keep state
pass inet proto udp from 192.168.103.0/24 to any port = ntp keep state
pass inet proto tcp from 192.0.2.227 to 192.168.103.0/24 port 9101:9103 flags S/SA keep state
```

Not bad, huh? Even with macros and lists expanded, the configuration is quite readable and you can follow the logic with no problems at all. If the requirements are those hosts that need to communicate with these services, this configuration give you what you need, and the default to block helps you keep away unwanted traffic.

Network Noise Removal, Revisited

But let's get back to keeping noise out of your network or at least your logs. Consider this (incomplete) configuration:

```
table <bruteforce> persist
block quick from <bruteforce>
pass inet proto tcp to $int_if:network port $tcp_services \
      keep state (max-src-conn 100, max-src-conn-rate 15/5, \
      overload <bruteforce> flush global)
```

The first line introduces the keyword table. PF's tables are data structures that are designed to store IP addresses. In a number of cases, you could specify lists of IP addresses as macros, but this in turn would transform your list into one rule per IP address in the loaded rule set. In contrast, the table is treated like one object in rule evaluation, and using `pfctl` subcommands, you can even change the contents of tables without reloading the PF configuration. Tables are useful in pass and block rules alike. If you write your pass rules to reference tables that correspond to specific groups of hosts in your network, adding access for a new host or removing access for a decommissioned one could be as easy as adding or deleting IP addresses from the table (see the References section at the end of the article for more material).

The other feature of this partial rule set that will catch your eye is the sequence of options in parentheses () after the keep state. These are state tracking options. The first, `max-src-conn`, specifies the maximum number of simultaneous connections from a single host. The second, `max-src-conn-rate`, specifies the maximum rate of new connections, here 15 over the run of 5 seconds. The third, `overload <bruteforce>`, specifies that any host that oversteps either of the limits will be added to that table, and finally, flush global means that all existing states for connections from a host that gets bumped to the overload table will be flushed from the state table. Traffic from hosts that overstep the limits will no longer match the pass rule, but we provided a block quick earlier in the rule set, and the offenders will be blocked by that rule when they retry. And since you were probably wondering, `$int_if:network` denotes the network interface name that the macro `$int_if` expands to and any network directly connected to that interface. You may want to use that notation to specify a local area network, for example.

Differentiate According to Services' and Users' Needs

There are several ways to tweak state tracking options to fit your specific needs; one obvious possibility is to introduce rules with different rate limits for specific kinds of traffic. The password guessing aimed at ssh and some other services has slowed down somewhat over the years, and a modern rule set might use something like this for ssh instead:

```
# tighter for ssh
pass quick proto tcp to port ssh \
    keep state (max-src-conn 15, max-src-conn-rate 5/3, \
    overload <bruteforce> flush global)
```

- most people can't type fast enough to manage five login attempts to the same host in the space of three seconds. Combinations like these are bound to save you a lot of noise in your authentication logs. It's also important to keep in mind that the tables may grow over time and that the compromised host that participated in some sort of unwanted access attempt last week may have been cleaned up afterwards. For that reason it is likely useful to expire table contents (using pfctl -T expire, see the man page) at intervals. Overflow tables are useful, but try as one might, rules like these are not entirely suited to deal with the distributed slow moving password guessers that started turning up some time in the 2005 to 2008 time frame.

On the Horizon: Your Environment, the PF Remix

PF in FreeBSD (and the somewhat further developed version in the original OpenBSD—for various somewhat understandable reasons, FreeBSD 10 ships with a PF that by now lags the OpenBSD source by more than six years) offers a friendly interface to a number of useful features and quite a few possibilities of sophisticated configuration for your systems. PF and related tools can help bring you load balancing, host and service redundancy (in essence, clustering functionality), traffic shaping, and even a few lightweight and effective spam-fighting features.

The configurations examples in this article are vaguely based on material from my tutorials and versions of the manuscript that evolved into *The Book of PF*, which is now available in its third edition. For further reading, you may want to start with the PF part of the *FreeBSD Handbook*'s Firewalls chapter (contributed by me, but further developed by various doc committers), *The Book of PF*, and of course the `pf.conf(5)` and `pfctl(8)` man pages as well as any references in those documents. If you want to study the code itself, it's available where you expect it to be. ●

REFERENCES

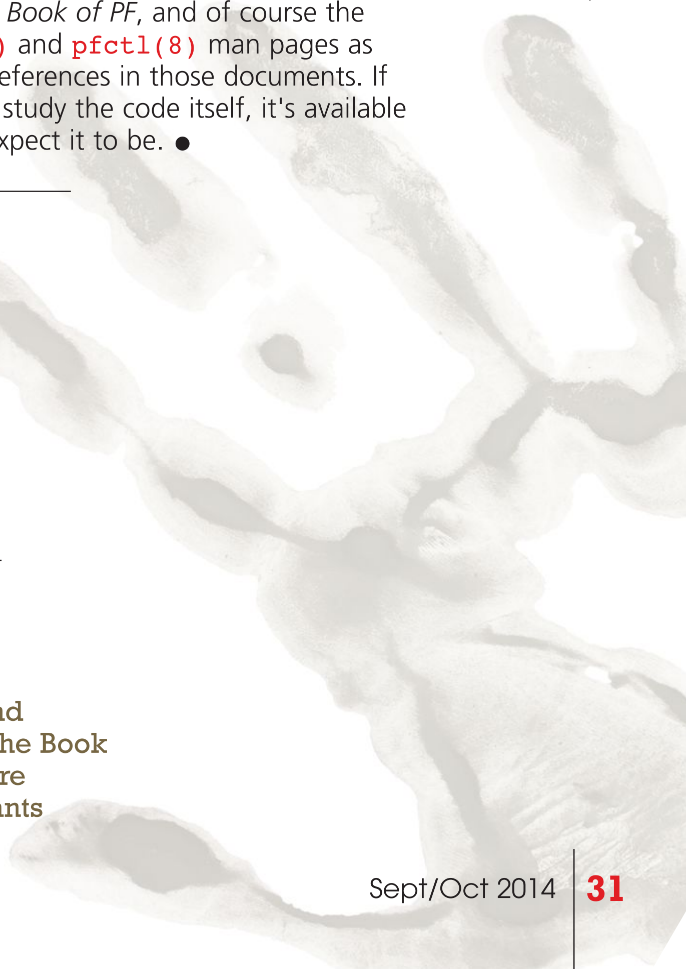
•Hansteen, Peter N. M. ***The Book of PF***. No Starch Press. (3rd edition 2014) http://www.nostarch.com/pf3

•Hansteen, Peter, Peter N. M. ***The Hail Mary Cloud and the Lessons Learned*** (blog post). (2013) (blog post) http://bsdly.blogspot.com/2013/10/the-hail-mary-cloud-and-lessons-learned.html

•***FreeBSD Handbook***, PF chapter. (1995–2014) https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/firewalls-pf.html

•***The PF man*** pages `pfctl(8) pf.conf(5)`

Peter N. M. Hansteen is a consultant, writer and sysadmin based in Bergen, Norway. He has been tinkering with computers since the mid 1980s, and rediscovered Unixes about the time 386BSD appeared. During recent years a frequent lecturer and tutor with emphasis on OpenBSD and FreeBSD topics, he has written a number of articles and The Book of PF (3rd edition No Starch Press 2014). In his blogosphere presence at http://bsdly.blogspot.com, he occasionally rants about (lack of) sanity in IT and related topics.



PORTSreport

by Frederic Culot

Our monthly dashboard showed a decrease in activity during this summer due to well-deserved holidays for our beloved committers. However, activity on the FreeBSD ports tree never stops. Here are the important events that happened during the last few months.

NEW PORTS COMMITTERS

It is a delight to welcome three new talents to the rank of ports committers.

- First, **Rui Paulo** (known as rpaulo@, as he already had a src commit bit) was awarded his ports commit bit after having done quite some work on the tree. It is always nice to see people from other areas (be they src or doc) join the rank of ports committers, as they often bring a fresh view to our processes.
- Then **Alonso Schaich** joined us and will be mentored by rakuco@ and makc@.
- And last but not least, **Dan Langille** (who people may already know through the FreeBSD Diary or FreshPorts, and as one of BSDCan's organizers) also received his commit bit.

IMPORTANT MILESTONES

On the 1st of September it will be time to bid farewell to the old pkg_ tools (see [http://blogs.freebsdish.org/portmgr/2014/02/03/time-to-bid-farewell-to-the-old-pkg_-tools/] for the long story). After years of service they will be replaced by the new and shiny pkg(8). Together with this switch, all remaining unstaged ports will be removed, which will open the door to better in-tree QA and many exciting things! Prepare yourself for great new features such as the ability to build and package as a non-root user and to package without installing anything, or the ability to create sub-packages. The future is bright for the ports tree!

NEW PACKAGE REPOSITORY

As recently announced on the freebsd-ports mailing list [https://lists.freebsd.org/pipermail/freebsd-ports/2014-August/094787.html], a new repository was created with packages built with support for stack protection (that is the -f stack-protector compilation flag is on). This is an important step towards better security when using software from the ports tree, and all our users are welcome to install and try packages from this repository before it becomes the default. To update your own packages with those that offer a better protection against buffer overflows, you just have to indicate the new SSP repository as mentioned in the initial announcement [https://lists.freebsd.org/pipermail/freebsd-security/2014-August/007874.html], and update your packages the usual way using pkg upgrade.

PORTS TREE 20TH ANNIVERSARY COMMEMORATION

As announced in our previous column, the 20th anniversary of the ports tree was celebrated on the 21st of August and a video was put online [http://youtu.be/LiFq5D-zmBs] for this occasion. We were pleased to see that the video was seen by about 2,000 people in two days!

Frederic Culot has worked in the IT industry for the past 10 years. During his spare time he studies business and management and just completed an MBA. Frederic joined FreeBSD in 2010 as a ports committer, and since then has made around 2,000 commits, mentored six new committers, and now assumes the responsibilities of portmgr-secretary.

NOTEWORTHY

CHANGES TO THE PORTS TREE

Despite the summer holidays, there were some changes that are worth mentioning here. Among them, the quarterly branch 2014Q3 was branched, the pkg tools were updated to reach version 1.3.7, and a lot of cleaning took place, mainly related to avoiding the usage of texinfo and libreadline from base, and to deal with unstaged ports.

svn UPDATE

by Glen Barber

IT IS THAT TIME OF THE YEAR AGAIN, and the FreeBSD 10.1-RELEASE cycle has begun. This column covers some of the new and exciting features expected to premier in FreeBSD 10.1.

Enhancements to bhyve

bhyve(8), a virtual machine hypervisor that runs natively on FreeBSD bhyve(8) first appeared in FreeBSD 10.0, and has been updated with many enhancements and features since then.

—stable/10@r267399—

bhyve(8) in FreeBSD 10.1 will now support 32-bit FreeBSD guest virtual machines, whereas the version in FreeBSD 10.0 only supports 64-bit guest machines.

—stable/10@r268932—

bhyve(8) now supports booting FreeBSD virtual machines that use a full ZFS file system.

—stable/10@r261090—

Support for the ACPI S5 state has been added, allowing guest machines to use the "soft power-off" ACPI feature.

Expanded ARM Platform Support

FreeBSD 10.1 builds upon the ARM platform support that was introduced in FreeBSD 10.1, adding new support for a number of systems, kernel updates, and userland updates.

In addition to the following referenced commits, support was added for the following boards:

- **CHROMEBOOK**
- **COLIBRI**
- **COSMIC**
- **IMX53-QSB**
- **QUARTZ**
- **RADXA**
- **WANDBOARD**

Additionally, symmetric multiprocessing support has been added, and all kernel configuration files for SMP-capable boards now have SMP enabled by default.

—stable/10@r266000—

Boot devices may now be specified by setting the 'loaderdev' u-boot environment variable, such as 'loaderdev=/dev/mmcsd0p1'. If a boot device is not specified in the u-boot environment, a fallback to the device probe mechanism will be used to determine the boot device.

UEFI Support

Changes to the FreeBSD/amd64 loader have been merged from FreeBSD-Current providing support for UEFI-boot. FreeBSD 10.1-RELEASE will support both UEFI and BIOS boot mechanisms, which will be available for the 10.1 release as separate ISO and memory stick images. Future releases are planned to combine both UEFI and BIOS boot support into a single image.

A significant portion of UEFI support development was sponsored by the FreeBSD Foundation.

New Automount Facility - autofs(5)

The new files ystem automount facility, autofs(5), has been merged from FreeBSD-Current. The new automounter is similar to that found in other UNIX-like operating systems, such as Apple OS X and Solaris, and uses a standardized, Sun-compatible configuration file.

Development of the autofs(5) implementation was sponsored by the FreeBSD Foundation.

New System Console Driver - vt(4)

The vt(4) driver, originally debuting in 9.3-RELEASE, is now available by default without requiring rebuilding the kernel. The vt(4) driver is the system console driver that integrates with KMS (Kernel Mode Setting) graphics cards.

Prior to the vt(4) driver, it would not have been possible to switch back to the system virtual terminals after the Xorg environment starts. This change adds a new loader(8) tunable, kern.vty, which activates the vt(4) virtual terminal driver when set to 'vt'.

The vt(4) driver has received a large number of updates and enhancements since FreeBSD 9.3-RELEASE, and support for the new system console driver has been added to a number of non-x86 architectures, such as powerpc64.

Much of the development on vt(4), as well as performance improvements and updates since the FreeBSD 9.3-RELEASE, was sponsored by the FreeBSD Foundation.

As a hobbyist, Glen Barber became heavily involved with the FreeBSD project around 2007. Since then, he has been involved with various functions, and his latest roles have allowed him to focus on systems administration and release engineering in the Project. Glen lives in Pennsylvania, USA.

this month In FreeBSD

BY DRU LAVIGNE

Several times per year, the FreeBSD Project publishes a Status Report that provides an overview of recent work. Since the status updates are written by the people actually doing the work, Status Reports provide unique insight into new features as they are developed, as well as a view to how the work is progressing and what still needs to be done. Status Reports also provide a picture of where the Project is going and reflect the quantity, quality, and diversity of work that occurs within the Project. Reports are written by developers, application porters, documentation writers and translators, and members of the infrastructure, core, security, and release engineering teams.

The first Status Report, published in June 2001, described their purpose as follows:

> One of the benefits of the FreeBSD development model is a focus on centralized design and implementation, in which the operating system is maintained in a central repository, and discussed on centrally maintained lists. This allows for a high level of coordination between authors of various components of the system, and allows policies to be enforced over the entire system, covering issues ranging from architecture to style. However, as the FreeBSD developer community has grown, the rate of both mailing list traffic and tree modifications has increased, making it difficult even for the most dedicated developer to remain on top of all the work going on in the tree.
>
> The FreeBSD Monthly Development Status Report attempts to address this problem by providing a vehicle that allows developers to make the broader community aware of their ongoing work on FreeBSD, both in and out of the central source repository. This is the first issue, and as such is an experiment. For each project and subproject, a one paragraph summary is included, indicating progress since the last summary (in this case, simply recent progress, as there have been no prior summaries).

This month, let's take a look at some of the work that was happening at the time of the first Status Report, as well as during the last quarter of the year 10 years ago, 5 years ago, and 1 year ago.

June 2001

The first Status Report contained 23 entries. Some of the notable features being worked on that are still used by FreeBSD users today were:
- A secure mechanism for the distribution of binary updates for FreeBSD
- Java port and licensed binary
- TrustedBSD work such as ACLs, auditing, and MAC

Q4 2004

By 2004, Status Reports also included status information about the FreeBSD Ports Collection and an update from the release engineering team. By Q4 2004:
- the Ports Collection contained over 12,000 ports
- at long last, FreeBSD 5.3 was released in November of 2004

This report contained 44 entries. Some highlights included:
- CARP ported from OpenBSD
- updates from the Freesbie and Frenzy live CD projects
- the portsnap and freebsd-update utilities released to provide for secure updates
- automation of Hardware Notes creation
- netperf work to enhance the performance of the FreeBSD networking stack
- SMPng work
- TCP cleanup and optimizations
- porting of the OpenBSD PF firewall into base

Q4 2009

This report contained 38 entries. By now:
- the Ports Collection contained over 21,000 ports
- FreeBSD 8.0 was released on November 26th, 2009

Some of the projects being worked on included:
- 3G USB support
- clang work
- HAST project to provide synchronous replication of any GEOM provider over a TCP/IP network
- SUJ (journaled soft updates)
- webcamd for enabling hundreds of different USB based webcam devices and V4L (Video for Linux) support in the Linux emulator
- wireless mesh networking (802.11s)
- new CAM based ATA implementation
- native NFSv4 ACL support in ZFS and UFS
- Flattened Device Tree (FDT) support

Q4 2013

This report contained 37 entries. By the last quarter of 2013:
- the Ports Collection contained approximately 24,500 ports
- the FreeBSD Release Engineering Team was finishing the 10.0-RELEASE cycle

And these were some of the projects being worked on:
- testing Jenkins and bhyve within the FreeBSD cluster to provide continuous integration and build testing
- native iSCSI stack
- UEFI boot
- new automounter (autofs)
- newcons (vt) replacement for virtual terminal emulator (syscons)
- work to enable FreeBSD as a fully supported compute host for OpenStack, using OpenContrail virtualized networking
- porting work to Cubieboard, Freescale i.MX6 processors, Freescale Vybrid VF6xx, and newer ARM boards
- Capsicum and Casper
- panicmail for centralized panic reporting
- testing the Kyua test suite LLDP debugger

The Project's next Status Report will be published in October 2014. All the Project's Status Reports can be accessed from https://www.freebsd.org/news/status/status.html.

Dru Lavigne is Director of the FreeBSD Foundation and Chair of the BSD Certification Group.

conference REPORT

by Daniel Peyrolón

EuroBSDcon 2014

EuroBSDcon, the premier European conference on open-source BSD operating systems, attracts highly-skilled engineering professionals, software developers, computer science students, professors, and users from Europe and other parts of the world. The goal of the conference is to exchange knowledge about the BSD operating systems and facilitate coordination and cooperation among users and developers. The main conference, which took place on September 27 and 28, 2014 in Sofia, Bulgaria, was ushered in by a two-day-long FreeBSD developer summit and tutorials.

The devsummit was great! As always, developers benefitted from the face-to-face contact and discussions, plus they enjoyed the opportunity to participate in decisions about the future of FreeBSD. There was also an excellent demonstration of tools. One example from that presentation that stands out in my mind is Phabricator, a tool that will prove to be a great help with code review for the project. And there was a discussion of package manager, which will be further improved for pkg 1.4—with one such improvement permitting complete granularity when installing the base system as a collection of packages.

Of great interest to attendees was the fact that FreeBSD is going to push the development of embedded architectures—like ARM and MIPS—with the objective of moving both architectures to Tier 1. Advancements have been made with upstreaming ASLR patches from HardenedBSD to enable ASLR to work on ARM architecture and to begin benchmarking it. The documentation team is also working to detect the weakest spots, so that they can further improve the overall quality of the documentation.

After the devsummit, it was time for the conference. I should start by saying that I particularly loved the fact that all major BSDs took part.

The starting shot of the conference was the keynote. FreeBSD Ports Collection creator Jordan Hubbard presented a talk about the future of FreeBSD—for the next 20 years—and the ways it can be improved to make it even more competitive in the market.

The topic of security was of great interest on both the FreeBSD and OpenBSD side, with presentations on ASLR implementation for FreeBSD, LibreSSL, using OpenBSD to test correctness of software, OpenBSD's arc4random implementation, and OpenBSD lazy binding implementation. And there were some very good talks on comparatively exotic projects from the NetBSD people. Some things weren't so exotic—like JIT compiled bpf—but others certainly were—like NPF scripting with Lua, or a talk given by Andy Tanembaum on Minix, which turns out to be utilizing NetBSD's user space. There were obviously a lot more talks presented at the conference--a clear sign of the strong momentum of BSD.

Interestingly, on the last day of the conference, George Neville-Neil and Sean Bruno managed to boot FreeBSD on the Linilo, a cheap embedded MIPS system. And at the conclusion of the conference, there was a talk by Atanas Chobanov about the anonymous submission of documents. Following that, the FreeBSD Foundation raffled a copy of *The Design and Implementation of the FreeBSD Operating System* (2nd edition), and Google raffled a Chromebook.

Next year EuroBSDcon will be held in Stockholm, Sweden, and if this year's conference is an indicator, it will be well worth attending. Start planning now! ●

Daniel Peyrolón is a master's degree student focusing on research in HPC. For the past two years he has also participated in the GSoC program, programming for the FreeBSD Project.

WELCOME
to AsiaBSDCon 2015!

DATE March 12-15, 2015

LOCATION

Tokyo University of Science,
Tokyo, Japan
www.http://asiabsdcon.org

CONTACT secretary@asiabsdcon.org

AUTHOR SCHEDULE

November 21, 2014..Deadline for Submission of Tutorial Proposals

November 28, 2014........................Deadline for Paper Submission

December 19, 2014...Notification of Acceptance of Papers and Tutorial Proposals

January 9, 2015....................................Deadline for Submission of Tutorial Materials

January 16, 2015 ....2015 Deadline for Submission of Final Papers

THROUGH DECEMBER

BY DRU LAVIGNE

2014 Events Calendar

The following BSD-related conferences are scheduled for the last quarter of 2014. More information about these events, as well as local user group meetings, can be found at **bsdevents.org.**

Ohio LinuxFest • Oct. 24–26, 2014 Columbus, OH

https://ohiolinux.org/ • There will be a FreeBSD booth and several FreeBSD-related talks at the 12th annual Ohio LinuxFest. The BSDA certification exam will also be available on Oct. 25 and 26.

MeetBSD California • Nov. 1–2 San Jose, CA

https://meetbsd.com • This biennial conference returns to the San Francisco Bay area. It uses a mixed, unConference format that includes scheduled presentations as well as BoFs, lightning talks, and speed geeking sessions. The BSDA certification exam will also be available during this event.

Silicon Valley Vendor & Developer Summits • Nov. 3–4 San Jose, CA

https://wiki.freebsd.org/201411VendorSummit • The annual Silicon Valley Vendor Summit has been expanded to two days to include developer summit sessions, resulting in a combined Developer/Vendor Summit. The full agenda and information on attending is available online.

OpenZFS DevSummit • Nov. 10–11 San Francisco, CA

http://open-zfs.org/wiki/OpenZFS_Developer_Summit_2014 • The goal of the second annual OpenZFS Developer Summit is to foster cross-community discussions of OpenZFS and its progress. The first day will consist of presentations and the second day will provide a hackathon.

LISA • Nov. 9–14 Seattle, WA

https://www.usenix.org/conference/lisa14 • The 28th Large Installation System Administration Conference will be held in Seattle. There will be a FreeBSD booth in the Expo area. Expo hours are 12:00-19:00 on Wednesday, November 12 and 10:00-14:00 on Thursday, November 13. Free expo-only passes are available from the conference website.

Are you aware of a conference, event, or happening that might be of interest to *FreeBSD Journal* readers?

Submit calendar entries to editor@freebsdjournal.com

FreeBSD™ JOURNAL

LEAK THIS!

• BLOG IT • TWEET IT • READ IT

amazon apps Available at amazon
Download on the App Store
ANDROID APP ON Google play

FreeBSD Journal is available as a mobile app for iPad, iPhone, Kindle, and all Android devices at $19.99 per year (for 6 issues), or $6.99 per single copy. ***NEW!*** Subscribe to the Desktop Edition at www.freebsdfoundation.org

FreeBSD™ Journal is published by The FreeBSD Foundation